Operating System Support for Extensible Secure File Systems

A RESEARCH PROFICIENCY EXAM PRESENTED
BY

CHARLES P. WRIGHT
STONY BROOK UNIVERSITY

Technical Report FSL-04-02
May 2004

Abstract of the RPE

Operating System Support for Extensible Secure File Systems

by

Charles P. Wright

Stony Brook University

2004

Securing data is more important than ever, yet secure file systems still have not received wide use. There are five primary security considerations: confidentiality, integrity, availability, authenticity, and non-repudiation. In this paper we analyze file systems that improve confidentiality and availability. We chose to focus on these two aspects, because theft of information and denial-of-service are the two largest costs associated with attacks.

Cryptographic file systems improve the confidentiality of files, because cleartext information is never stored on disk. One barrier to the adoption of cryptographic file systems is that the performance impact is assumed to be too high, but in fact is largely unknown. In this paper we first survey available cryptographic file systems, and perform a performance comparison of a representative set of the systems, emphasizing multiprogrammed workloads. We show the overhead of cryptographic file systems can be minimal for many real-world workloads. We have observed not only general trends with each of the cryptographic file systems we compared but also anomalies based on complex interactions with the operating system, disks, CPUs, and ciphers.

Snapshotting and versioning systems increase availability, because before data is modified or deleted, a copy is made. This prevents an attacker from destroying valuable data. Indeed, snapshotting and versioning are useful beyond security applications. Users or administrators often make mistakes, which result in the destruction of data; snapshotting or versioning allow simple recovery from such mistakes. A useful extension of snapshotting is sandboxing: allowing suspect processes to modify a separate copy of the data, while the original data remains intact.

We then discuss operating system changes we made to support our encryption and snapshotting file systems. Finally, we conclude by suggesting future OS enhancements and potential improvements to existing cryptographic file systems.

To my family — old and new.

Contents

List of Figures

List of Tables

Acknowledgments

The largest thanks go to Mariya who has endured the countless days, nights, and weekends in the lab that went into this work. My advisor, Erez Zadok, has always been available, helpful, and without him I would not have been able to complete this research. I performed the benchmarking of cryptographic file systems with Jay Dave. Puja Gupta, Harikesavan Krishan, and Mohammad Nayyer Zubair wrote much of the original Unionfs code that became the basis for snapshotting and sandboxing. My other lab mates have also been great to work with. I would be remiss if I did not acknowledge Kiran Reddy who sat next to me debugging race conditions for months on end.

My committee members, R. Sekar and Tzi-cker Chiueh, were generous of their time and provided helpful input and comments on the work. The anonymous SISW and USENIX reviewers provided very useful comments that improved the quality of this work. Jim Hughes, Serge Plotkin, and others at the IEEE SISW workshop also provided valuable comments. Anton Altaparmakov and Szakacsits Szabolcs helped us to understand the behavior of NTFS.

This work was partially made possible thanks to NSF CAREER award EIA-0133589, NSF award CCR-0310493, and HP/Intel gifts numbers 87128 and 88415.1.

Chapter 1

Introduction

Securing data is more important than ever. As the Internet has become more pervasive, attacks have grown. Widely-available studies report millions of dollars of lost revenues due to security breaches [45, 51]. Moreover, the theft of proprietary information consistently caused the greatest financial loss. Data loss, either malicious or accidental, also costs millions of dollars [15, 30]. Backup companies report that on average, 20MB of data takes 30 hours to create and is worth almost $90,000 [3].

In this report we survey solutions to these two important problems in storage security. To protect confidentiality we investigate encryption file systems; and to protect data availability we investigate versioning and snapshotting systems. We then discuss operating system improvements that we have made to support our encryption and snapshotting file systems.

We believe there are three primary factors when evaluating encrypting file systems: security, performance, and ease-of-use. These concerns often compete; for example, if a system is too difficult to use, then users simply circumvent it entirely. Furthermore, if users perceive encryption to slow down their work, they just turn it off. Though performance is an important concern when evaluating cryptographic file systems, no rigorous real-world comparison of their performance has been done to date.

Riedel described a taxonomy for evaluating the performance and security that each type of system could theoretically provide [52]. His evaluation encompassed a broad array of choices, but because it was not practical to benchmark so many systems, he only drew theoretical conclusions. In practice, however, deployed systems interact with disks, caches, and a variety of other complex system components — all having a dramatic effect on performance.

We expect cryptographic file systems to become a commodity component of future operating systems. In this paper we perform a real world performance comparison between several systems that are used to secure file systems on laptops, workstations, and moderately-sized file servers. We also emphasize multiprogramming workloads, which are not often investigated. Multi-programmed workloads are becoming more important even for single user machines, in which Windowing systems are often used to run multiple applications concurrently.

We present results from a variety of benchmarks, analyzing the behavior of file systems for metadata operations, raw I/O operations, and combined with CPU intensive tasks. We also use a variety of hardware to ensure that our results are valid on a range of systems: Pentium vs. Itanium, single CPU vs. SMP, and IDE vs. SCSI. We observed general trends with each of the cryptographic file systems we compared. We also investigated anomalies due to complex, counterintuitive interactions with the operating system, disks, CPU, and ciphers. We propose future directions and enhancements to make systems more reliable and predictable.

We also believe that snapshotting and versioning systems are likely to become common OS components. We define snapshotting as the act of creating a consistent view of an entire file system, and versioning as making backup copies of individual files. Snapshotting systems are useful for system administrators and multi-user systems, they essentially provide on-line and instant backups that can protect against software errors and user actions. If an attacker modifies data, the system administrator can quickly revert to the latest snapshot, or easily find the changes made since the last snapshot.

Versioning, on the other hand, is useful to end users, because they are able to examine changes to their own files in detail. Versioning systems are also typically more fine-grained than a snapshotting system. Versions can be created on each write to a file, but having so many versions of a file system would have a large performance impact and be difficult for the system administrator to manage.

Snapshotting and versioning have uses beyond security. For example, system administrators need to know what changes are made to the system while installing new software. If the installation failed, the software does not work as advertised, or is not needed, then the administrator often wants to revert to a previous good system state. In this case a snapshotting system is ideal, because the entire file system state can easily be reverted to the preinstall snapshot. Simple mistakes such as the accidental modification of files could be ameliorated if users could simply execute a single command to undo such accidental changes. In this case, a versioning file system would allow the user to recover this individual file, without affecting other files (or other users' data).

The rest of this paper is organized as follows. Chapter 2 surveys cryptographic file systems and ciphers. Chapter 3 compares the performance of a cross section of cryptographic file systems. Chapter 4 surveys snapshotting and versioning file systems. Chapter 5 describes operating system changes we have made in support of our new file systems. We conclude in Chapter 6 and present possible future directions.

Chapter 2

Cryptographic File System Survey

This section describes a range of available techniques of encrypting data and is organized by increasing levels of abstraction: block-based systems, native disk file systems, network-based file systems, stackable file systems, and encryption applications. We conclude this section by discussing ciphers that are suitable for encrypted storage.

2.1 Block-Based Systems

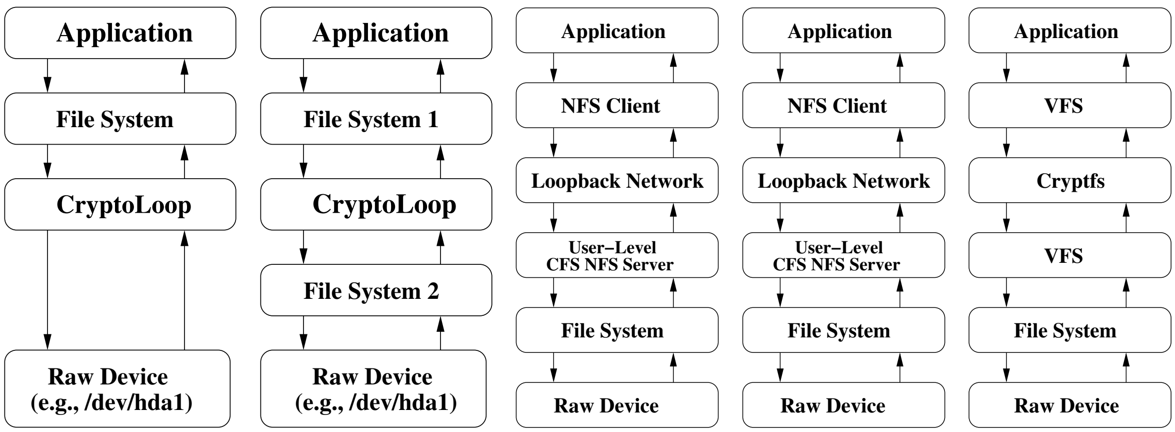
Block-based encryption systems operate below the file system level, encrypting one disk block at a time. This is advantageous because they do not require knowledge of the file system that resides on top of them, and can even be used for swap partitions or applications that require access to raw partitions (e.g., database servers). Also, they do not reveal information about individual files (e.g., sizes) or directory structure.

2.1.1 Cryptoloop

The Linux loopback device driver presents a file as a block device, optionally transforming the data before it is written and after it is read from the native file, usually to provide encryption. Linux kernels include a cryptographic framework, CryptoAPI [53] that exports a uniform interface for all ciphers and hashes. Presently, IPsec and the Cryptoloop driver use these facilities.

We investigated three backing stores for the loopback driver: (1) a preallocated file created using `dd` filled with random data, (2) a raw device (e.g., `/dev/hda2`), and (3) a sparse backing file created using `truncate(2)`. Figure 2.1(a) shows the path that data takes from an application to the file system, when using a raw device as a backing store. Figure 2.1(b) shows the path when a file is used as the backing store. The major difference between the two systems is that there is an additional file system between the application and the disk when using a file instead of a device. Using files rather than devices adds performance penalties including cutting the effective buffer cache in half because blocks are stored in memory both as encrypted and unencrypted data. Each of these methods has advantages and disadvantages related to security and ease-of-use as well. Using preallocated files is more secure than using sparse files, because an attacker can not distinguish random data in the file from encrypted data. However, to use a preallocated file, space must be set aside for encrypted files ahead of time. Using a sparse backing store means that space does not need to be preallocated for encrypted data, but reveals more about the structure of the file system (since the attacker knows where and how large the holes are). Using a raw device allows an entire disk or partition to be encrypted, but requires repartitioning, which is more complex than simply creating a file. Typically there are also a limited number of partitions available, so on multi-user systems encrypted raw devices are not as scalable as using files as the backing store.

In Linux 2.6.4, the Cryptoloop driver has been deprecated by dm-crypt. dm-crypt uses the generic device mapping API provided by 2.6 that is also used by the LVM subsystem. dm-crypt is similar to Cryptoloop, and the architectural overview remains the same. dm-crypt still uses the CryptoAPI, and the on-disk encryption format has remained the same. There are two key implementation advantages of dm-crypt over Cryptoloop: (1) encryption is separated from the loopback functionality, and (2) dm-crypt uses memory pools to prevent memory allocation from within the block device driver (eliminating deadlocks under memory pressure).



(a) Cryptoloop (raw device) (b) Cryptoloop (file) (c) CFS (d) TCFS (e) Stackable file systems

Figure 2.1: Call paths of storage encryption systems.

2.1.2 CGD

The CryptoGraphic Disk driver, available in NetBSD, is similar to the Linux loopback device, and other loop-device encryption systems, but it uses a native disk or partition as the backing store [13]. CGD has a fully featured suite of user-space configuration utilities that include n-factor authentication and PKCS#5 for transforming user passwords into encryption keys [54]. This system is similar to Cryptoloop using a raw device as a backing store.

2.1.3 GBDE

GEOM-based disk encryption (GBDE) is based on GEOM, which provides a modular framework to perform transformations ranging from simple geometric displacement for disk partitioning, to RAID algorithms, to cryptographic protection of the stored data. GBDE is a GEOM transform that enables encrypting an entire disk [28]. GBDE hashes a user-supplied passphrase into 512 bits of key material. GBDE uses the key material to locate and encrypt a 2048 bit master key and other metadata on four distinct *lock sectors*. When an individual sector is encrypted, the sector number and bits of the master key are hashed together using MD5 to create a *kkey*. A randomly generated key, the *sector key*, is encrypted with the *kkey*, and then written to disk. Finally, the sector's payload is encrypted with the sector key and written to disk. This technique, though more complex, is similar to Cryptoloop using a raw device as a backing store.

2.1.4 SFS

SFS is an MSDOS device driver that encrypts an entire partition [21]. SFS is similar to Cryptoloop using a raw device as a backing store. Once encrypted, the driver presents a decrypted view of the encrypted data. This provides the convenient abstraction of a file system, but relying on MSDOS is risky because MSDOS provides none of the protections of a modern operating system.

2.1.5 BestCrypt

BestCrypt is a commercially available loopback device driver supporting many ciphers [27]. BestCrypt supports both Linux and Windows, and uses a normal file as a backing store (similar to using a preallocated file with Cryptoloop).

2.1.6 vncrypt

vncrypt is the cryptographic disk driver for FreeBSD, based on the *vn*(4) driver that provides a disk-like interface to a file. vncrypt uses a normal file as a backing store (similarly to using a preallocated file with Cryptoloop) and provides a character device interface, which FreeBSD uses for file systems and swap devices. vncrypt is similar to using Cryptoloop with a file as a backing store.

2.1.7 OpenBSD vnd

Encrypted file systems support is part of OpenBSD's Vnode Disk Driver, *vnd*(4). *vnd* uses two modes: one bypasses the buffer cache, the second uses the buffer cache. For encrypted devices, the buffer cache must be used to ensure cache coherency on unmount. The only encryption algorithm implemented so far is Blowfish. *vnd* is similar to using Cryptoloop with a file as a backing store.

2.2 Disk-Based File Systems

Disk-based file systems that encrypt data are located at a higher level of abstraction than block-based systems. These file systems have access to all per-file and per-directory data, so they can perform more complex authorization and authentication than block-based systems, yet at the same time disk-based file systems can control the physical data layout. This means that disk-based file systems can limit the amount of information revealed to an attacker about file size and owner, though in practice these attributes are often still revealed in order to preserve the file system's on-disk structure. Additionally, since there is no additional layer of indirection, disk-based file systems can have performance benefits over other techniques described in this section (including the loop devices).

2.2.1 EFS

EFS is the Encryption File System found in Microsoft Windows, based on the NT kernel (Windows 2000 and XP) [37]. It is an extension to NTFS and utilizes Windows authentication methods as well as Windows ACLs [39, 52]. Though EFS is located in the kernel, it is tightly coupled with user-space DLLs to perform encryption and the user-space Local Security Authentication Server for authentication [61]. This prevents EFS from being used for protecting files or folders in the root or `\winnt` directory. Encryption keys are stored on the disk in a *lockbox* that is encrypted using the user's login password. This means that when users change their password, the lockbox must be re-encrypted. If an administrator changes the user's password, then all encrypted files become unreadable. Additionally, for compatibility with Windows 2000, EFS uses

DESX [57] by default and the only other available cipher is 3DES (included in Windows XP or in the Windows 2000 High Encryption pack).

2.2.2 StegFS

StegFS is a file system that employs steganography as well as encryption [35]. If adversaries inspect the system, then they only know that there is hidden data, but not its content or the extent of what is hidden. This is achieved via a modified Ext2 kernel driver that keeps a separate block-allocation table per *security level*. It is not possible to determine how many security levels exist without the key to each security level. When the disk is mounted with an unmodified Ext2 driver, random blocks may be overwritten, so data is replicated randomly throughout the disk to avoid data loss. Although StegFS achieves plausible deniability of data's existence, the performance degradation is a factor of 6–196, making it impractical for most applications.

2.3 Network Loopback Based Systems

Network-based file systems (NBFSs) operate at a higher level of abstraction than disk-based file systems, so NBFSs can not control the on-disk layout of files. NBFSs have two major advantages: (1) they can operate on top of any file system, and (2) they are more portable than disk-based file systems. NBFS's major disadvantages are performance and security. Since each request must travel through the network stack, more data copies are required and performance suffers. Security also suffers because NBFS are vulnerable to all of the weaknesses of the underlying network protocol (usually NFS [60, 70]).

2.3.1 CFS

CFS is a cryptographic file system that is implemented as a user-level NFS server [6]. The cipher and key are specified when encrypted directories are first created. The CFS daemon is responsible for providing the owner with access to the encrypted data via an *attach* command. The daemon, after verifying the user ID and key, creates a directory in the mount point directory that acts as an unencrypted window to the user's encrypted data. Once attached, the user accesses the attached directory like any other directory. CFS is a carefully designed, portable file system with a wide choice of built-in ciphers. Its main problem, however, is performance. Since it runs in user mode, it must perform many context switches and data copies between kernel and user space. As can be seen in Figure 2.1(c), since CFS has an unmodified NFS client which communicates with a modified NFS server, it must run only over the loopback network interface, *lo*.

2.3.2 TCFS

TCFS is a cryptographic file system that is implemented as a modified kernel-mode NFS client. Since it is used in conjunction with an NFS server, TCFS works transparently with the remote file system. To encrypt data, a user sets an encrypted attribute on directories and files within the NFS mount point [8, 34]. TCFS integrates with the UNIX authentication system in lieu of requiring separate passphrases. It uses a database in `/etc/tcfspwdb` to store encrypted user and group keys. Group access to encrypted resources is limited to a subset of the members of a given UNIX group, while allowing for a mechanism (called *threshold secret sharing*) for reconstructing a group key when a member of a group is no longer available. As can be seen on the right of Figure 2.1(d), TCFS uses a modified NFS client, which must be implemented in the kernel. This does, however, allow it to operate over any network interface and to work with remote servers. Since TCFS runs on individual clients, and the NFS server is unmodified this increases scalability because cryptographic processing takes place on the client instead of the server.

TCFS has several weaknesses that make it less than ideal for deployment. First, the reliance on login passwords as user keys is not safe. Also, storing encryption keys on disk in a key database further reduces security. Finally, TCFS operates only on systems with Linux kernel 2.2.17 or earlier.

2.4 Stackable File Systems

Stackable file systems are a compromise between kernel-level disk-based file systems and loopback network file systems. Stackable file systems can operate on top of any file system; do not copy data across the user-kernel boundary or through the network stack; and they are portable to several operating systems [73].

2.4.1 Cryptfs

Cryptfs is the stackable, cryptographic file system and part of the FiST toolkit [72]. Cryptfs was never designed to be a secure file system, but rather a proof-of-concept application of FiST [73]. Cryptfs supports only one cipher and implements a limited key management scheme. Cryptfs serves as the basis for several commercial and research systems (e.g., ZIA [11]). Figure 2.1(e) shows Cryptfs's operation. A user application invokes a system call through the Virtual File System (VFS). The VFS calls the stackable file system, which again invokes the VFS after encrypting or decrypting the data. The VFS calls the lower level file system, which writes the data to its backing store.

2.4.2 NCryptfs

NCryptfs is our stackable cryptographic file system, designed with the explicit aim of balancing security, convenience, and performance [69]. NCryptfs allows system administrators and users to customize NCryptfs according to their specific needs. NCryptfs supports multiple concurrent authentication methods, multiple dynamically-loadable ciphers, ad-hoc groups, and challenge-response authentication. Keys, active sessions, and authorizations in NCryptfs all have timeouts. NCryptfs can be configured to transparently suspend and resume processes based on key, session or authorization validity. NCryptfs also enhanced the kernel to discard cleartext pages and notify the file system on process exit in order to expunge invalid authentication entries.

2.4.3 ZIA

Zero-Interaction Authentication (ZIA) uses Cryptfs with Rijndael as a basis for a system that uses physical locality to authorize use on a laptop [11]. To effectively use file system encryption, users must provide the system with a passphrase. Normally, after the user provides the machine with the passphrase, anyone who physically possess the machine has access to the data. The laptop running ZIA communicates with an ancillary device that is always in the physical possession of the user, (e.g., a wrist-watch) that provides the laptop with the encryption key over a wireless channel. When the device leaves the vicinity of the laptop, all information in the file cache is encrypted. When the user returns, the information is automatically decrypted.

2.5 Applications

File encryption can be performed by applications such as GPG [31] or crypt(1) that reside above the file system. This solution, however, is quite inconvenient for users. Each time they want to access a file, users must manually decrypt or encrypt it. The more user interaction is required to encrypt or decrypt files, the more often mistakes are made, resulting in damage to the file or leaking confidential data [67]. Additionally, the file may reside in cleartext on disk while the user is actively working on it.

File encryption can also be integrated into each application (e.g., mail clients), but this shifts the burden from users to applications programmers. Often applications developers do not believe the extra effort of implementing features is justified when only a small fraction of users would take advantage of those features. Even if encryption is deemed an important enough feature to be integrated into most applications there are still two major problems with this approach. First, each additional application that the user must trust to function correctly reduces the overall security of the system. Second, since each application may implement encryption slightly differently it would make using files in separate programs more difficult.

2.6 Ciphers

For cryptographic file systems there are several ciphers that may be used, but those of interest are generally symmetric block ciphers. This is because block ciphers are efficient and versatile. We discuss DES variants, Blowfish, and Rijndael because they are often used for file encryption, and are believed to be secure. There are many other block ciphers available, including CAST, GOST, IDEA, MARS, Serpent, RC5, RC6, and TwoFish. Most of them have similar characteristics with varying block and key sizes. We also discuss tweakable ciphers, a new cipher mode which is especially suited for encrypted storage.

2.6.1 DES

DES is a block cipher designed by IBM with assistance from the NSA in the 1970s [57]. DES was the first encryption algorithm that was published as a standard by NIST with enough details to be implemented in software. DES uses a 56-bit key, a 64-bit block size, and can be implemented efficiently in hardware. DES is no longer considered to be secure. There are several more secure variants of DES, most commonly 3DES [40]. 3DES uses three separate DES encryptions with three different keys, increasing the total key length to 168 bits. 3DES is considered secure for government communications. DESX is a variant designed by RSA Data Security that uses a second 64-bit key for *whitening* (obscuring) the data before the first round and after the last round of DES proper, thereby reducing its vulnerability to brute-force attacks, as well as differential and linear cryptanalysis [57].

2.6.2 Blowfish

Blowfish is a block cipher designed by Bruce Schneier with a 64-bit block size and key sizes up to 448 bits [57]. Blowfish had four design criteria: speed, compact memory usage, simple operations, and variable security. Blowfish works best when the key does not change often, as is the case with file encryption, because the key setup routines require 521 iterations of Blowfish encryption. Blowfish is widely used for file encryption.

2.6.3 AES (Rijndael)

AES is the successor to DES, selected by a public competition. Though all of the six finalists were judged to be sufficiently secure for AES, the final choice for AES was Rijndael based on the composite of the three selection criteria (security, cost, and algorithm characteristics) [41]. Since AES has been blessed by the US government and government purchases will require AES encryption, it is likely that AES will become more popular. Rijndael is a block cipher based on the Square cipher that uses S-boxes (substitution), shifting, and XOR to encrypt 128-bit blocks of data. Rijndael supports 128, 192, and 256 bit keys.

2.6.4 Tweakable Ciphers

Tweakable block ciphers are new cipher mode (other cipher modes include electronic codebook (ECB), cipher block chaining (CBC), and cipher feedback (CFB)) [22]. Tweakable block ciphers operate on large block sizes (e.g., 512-bytes) and are ideal for sector level-encryption. The IEEE Security in Storage Working Group is planning on using the AES encrypt-mix-encrypt (EME) cipher mode for their project 1619 "Standard Architecture for Encrypted Shared Storage Media."

As can be seen in Figure 2.2, encrypt-mix-encrypt essentially divides the sector into encryption-block–size units ($P_1, P_2, \ldots, P_n$), encrypts each one, mixes it with the result of all the first round encryptions, and then encrypts the result. Specifically, each P_i is encrypted once with AES, and then all the results are exclusive-ored to create M. Tweakable cipher modes do not use an IV, but rather use a tweak T (T is not directly shown in the figure, but is a component of M). M is then calculated as $M \oplus T$. The result from the previous encryptions are exclusive-ored with M, and then encrypted with AES again. Each block is additionally exclusive-ored with iL (a constant derived from $E_K(0)$), before the first encryption round and after the last encryption round. In sum, $C_i = iL \oplus E_K(E_K(iL \oplus P_i) \oplus M)$, where $M = E_K(L \oplus P_1) \oplus \ldots E_K(nL \oplus P_n) \oplus T$.

Unlike CBC, tweakable ciphers have very large error propagation. Since each output block includes the tweak (presumably based on the sector) and random information from the first encryption round, small modifications are propagated throughout the sector. If there are any modifications to the sector, then they will randomly affect each bit of the sector. The large error propagation has two advantages: (1) even a small change to data is more likely to be noticed, and (2) it is not susceptible to CBC cut-and-paste attacks.

EME ciphers are also more suited to hardware acceleration than standard CBC mode. In CBC mode to encrypt the i-th data block, the result from the $i-1$-th block is required, so to encrypt n blocks, there is a delay of n units. However, in EME mode only one synchronization point is needed (to compute M) and only two encryption delays are required (one for the first round and one for the second round).

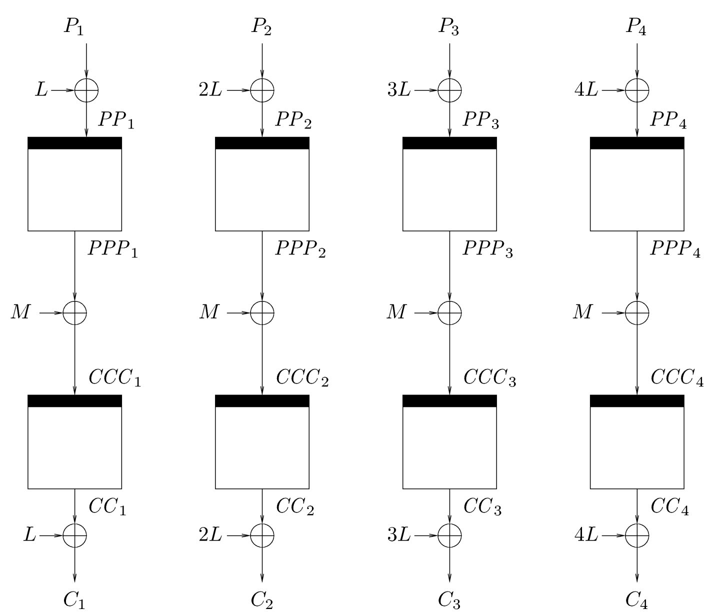


Figure 2.2: $L = E_K(O)$ and $M = (PPP_1 \oplus \ldots PPP_n) \oplus T$, where T is the tweak. Figure taken from Halevi and Rogaway [22].

Chapter 3

Encryption File System Performance Comparison

To compare the performance of cryptographic systems we chose one system from each category in Section 2, because we were interested in evaluating properties of techniques, not specific systems. We chose benchmarks that measure file system operations, raw I/O operations, and a simulated user workload. Throughout the comparisons we ran multi-programmed tests to compare increasingly parallel workloads and scalability.

We did not benchmark an encryption application, because other programs can not transparently access encrypted data. As representatives from each category we chose Cryptoloop, EFS, CFS, and NCryptfs. We chose Cryptoloop, EFS, and CFS because they are widely used, and up-to-date. Cryptoloop can be run on top of raw devices and normal files, which makes it comparable to a wider range of block-based systems than a block-based system that uses only files or only block devices. We chose NCryptfs over Cryptfs because NCryptfs is more secure. We also tried to choose systems that run on a single operating system, so that operating system effects would be consistent. We used Linux for most systems, but for disk-based file systems we chose Windows. There was no suitable and widely-used disk-based solution for Linux, large part because block-based systems are generally used on Linux. From previous studies, we also knew that TCFS and BestCrypt had performance problems and we therefore omitted them [69].

We chose workloads that stressed file system and I/O operations, particularly when there are multiple users. We did not use a compile benchmark or other similar workloads, because they do not effectively measure file system performance under heavy loads [65]. In Section 3.1 we describe our experimental setup. In Section 3.2 we report on PostMark, a benchmark that exercises file system meta-data operations such as lookup, create, delete, and append. In Section 3.3 we report on PGMeter, a benchmark that exercises raw I/O throughput. In Section 3.4 we report on AIM Suite VII, a benchmark that simulates large multiuser workloads. Finally, in Section 3.5 we report other interesting results.

3.1 Experimental Setup

To provide a basis for comparison we also perform benchmarks on Ext2 and NTFS. Ext2 is the baseline for Cryptoloop, CFS, and NCryptfs. NTFS is the baseline for EFS. We chose Ext2 rather than other Linux file systems because it is widely used and well-tested. We chose Ext2 rather than the Ext3 journaling file system, because the loopback device's interaction with Ext3 has the effect of disabling journaling. For performance reasons, the default journaling mode of Ext3 does not journal data, but rather only journals meta-data. When a loopback device is used with a backing store in an Ext3 file, the Ext3 file system contained inside the file does journal meta-data, but it writes the journal to the lower-level Ext3 file as data, which is not journaled. Additionally, journaling file systems create more complicated I/O effects by writing to the journal as well as normal file system data.

We used the following configurations:

- A vanilla Ext2 file system.
- A loopback device using a raw partition as a backing store. Ext2 is used inside the loop device. We refer to this configuration as LOOPDEV.
- A loopback device using a preallocated backing store. Both the underlying file system and the file system contained within the loop device are Ext2. We refer to this configuration as LOOPDD.
- CFS using an Ext2 file system as a backing store.
- A vanilla NTFS file system.
- An encrypted folder within an NTFS file system (EFS).
- NCryptfs stacked on top of an Ext2 file system.

3.1.1 System Setup

For Cryptoloop and NCryptfs we used four ciphers: the Null (identity) transformation to demonstrate the overhead of the technique without cryptography; Blowfish with a 128-bit key to demonstrate overhead with a cipher commonly used in file systems; AES with a 128-bit key because AES is the successor to DES [12]; and 3DES because it is an often-used cipher and it is the only cipher that all of the tested systems supported. We chose to use 128-bit keys for Blowfish and AES, because that is the default key size for many systems. For CFS we used Null, Blowfish, and 3DES (we did not use AES because CFS only supports 64-bit ciphers). For EFS we used only 3DES, since the only other available cipher is DESX.

	Feature	**LOOPDEV**	**LOOPDD**	**EFS**	**CFS**	**NCryptfs**
1	**Category**	Block Based	Block Based	Disk FS	NFS	Stackable FS
2	**Location**	Kernel	Kernel	Hybrid	User Space	Kernel
3	**Buffering**	Single	Double	Single	Double	Double
4	**Encryption unit**	512B	512B	512B	8B	4KB/16KB
5	**Encryption mode**	CBC	CBC	CBC (?)	OFB+ECB	CTS/CFB
6	**Write Mode**	Sync,Async	Async Only	Sync,Async	Async Only	Async only Write-Through

Table 3.1: Peformance related features.

In Table 3.1 we summarize various system features that affect performance.

1. Category LOOPDEV and LOOPDD use a block-based approach. EFS is implemented as an extension to the NTFS disk-based file system. CFS is implemented as a user-space NFS server. NCryptfs is a stackable file system.

2. Location LOOPDEV, LOOPDD, and NCryptfs are implemented in the kernel. EFS uses a hybrid approach, relying on user-space DLLs for some cryptographic functions. CFS is implemented in user space.

3. Buffering LOOPDEV and EFS keep only one copy of the data in memory. LOOPDD, CFS, and NCryptfs have both encrypted and decrypted data in memory. Double buffering effectively cuts the buffer cache size in half.

4. Encryption unit LOOPDEV, LOOPDD, and EFS use a disk block as their unit of encryption. This defaults to 512 bytes for our tested systems. CFS uses the cipher block size as its encryption unit and only supports ciphers with 64-bit blocks. NCryptfs uses the PAGE_SIZE as the unit of encryption: on the i386 this is 4KB and on the Itanium this defaults to 16KB.

5. Encryption mode LOOPDEV and LOOPDD use CBC encryption. The public literature does not state what mode of encryption EFS uses, though it is most probably CBC mode because: (1) the Microsoft CryptoAPI uses CBC mode by default, and (2) it has a fixed block size that is accommodating to CBC. NCryptfs uses *cipher text stealing* (CTS) to encrypt data of arbitrary length to data of the same length.

CFS does not use standard cipher modes, but rather a hybrid of ECB and OFB. Chaining modes (e.g., CBC) do not allow direct random access to files, because to read or write from the middle of a file all the previous data must be decrypted or encrypted first. However, ECB mode permits a cryptanalyst to do structural analysis. CFS solves this by doing the following: when an attach is created, half a megabyte of pseudo-random data is generated using OFB mode and written to a *mask file*. Before data is written to a data file, it is XORed with the contents of the mask file at the same offset as the data (modulo the size of the mask file), then encrypted with ECB mode. To read data this process is simply reversed. This method is used to allow uniform access time to any portion of the file while preventing structural analysis. NCryptfs and Cryptoloop achieve the same effect using initialization vectors (IVs) and CBC on pages and blocks rather than the whole file.

For Cryptoloop, CFS, and NCryptfs we adapted the Blowfish, AES, and 3DES implementations from OpenSSL 0.9.7b to minimize effects of different cipher implementations [46]. Cryptoloop uses CBC mode encryption for all operations. NCryptfs uses CTS to ensure that an arbitrary length buffer can be encrypted to a buffer of the same length [57]. NCryptfs uses CTS because size changing algorithms add complexity and overhead to stackable file systems [71]. CTS mode differs from CBC mode only for the last two blocks of data. For buffers that are shorter than the block size of the cipher, NCryptfs uses CFB, since CTS requires at least one full block of data. We did not change the mode of encryption that CFS uses.

6. Write Mode LOOPDD does not use a synchronous file as its backing store, therefore all writes become asynchronous. CFS uses NFSv2 where all writes on the server must be synchronous, but CFS violates the specification by opening all files asynchronously. Due to VFS calling conventions, NCryptfs does not cause the lower-level file system to use write synchronously. Asynchronous writes improve performance, but at the expense of reliability. NCryptfs uses a write-through strategy: whenever a `write` system call is issued, NCryptfs passes it to the lower-level file system. The lower-level file system may not do any I/O, but NCryptfs still encrypts data.

3.1.2 Testbed

We ran our benchmarks on two machines: the first represents a workstation or a small work group file server, and the second represents a mid-range file server.

The workstation machine is a 1.7Ghz Pentium IV with 512MB of RAM. In this configuration all experiments took place on a 20GB 7200 RPM Western Digital Caviar IDE disk. For Cryptoloop, CFS, and NCryptfs we used Red Hat Linux 9 with a vanilla Linux 2.4.21 kernel. For EFS we used Windows XP (Service pack 1) with high encryption enabled for EFS.

Our file server machine is a 2 CPU 900Mhz Itanium 2 McKinley (hereafter we refer to this CPU as an Itanium) with 8GB of RAM running Red Hat Linux 2.1 Advanced Server with a vanilla SMP Linux 2.4.21 kernel. All experiments took place on a Seagate 73GB 10,000 RPM Cheetah Ultra160 SCSI disk. Only Ext2, Cryptoloop, and NCryptfs were tested on this configuration. We did not test CFS because its file handle and encryption code are not 64-bit safe. We did not test NTFS or EFS because 64-bit Windows is not yet commonly used on the Itanium platform.

All tests were performed on a cold cache, achieved by unmounting and remounting the file systems between iterations. The tests were located on a dedicated partition in the outer sectors of the disk to minimize ZCAV and other I/O effects [14].

For multi-process tests we report the elapsed time as the maximum elapsed time of all processes, which shows how long it takes to actually get the allotted work completed. We report the system time as the sum of the system times of each process and kernel thread involved in the test. For LOOPDD we add the CPU time of the kernel thread used by the loopback device to perform encryption. For LOOPDEV we add in the CPU time used by `kupdated`, because encryption takes place when syncing dirty buffers. Finally, for CFS we add the user and system time used by `cfsd`, because this is CPU time used on behalf of the process.

As expected there were no significant variations in the user time of the benchmark tools, because no changes took place in the user code. We do not report user times for these tests because we do not vary the amount of work done in the user process.

We ran all tests several times, and we report instances where our computed standard deviations were more than 5% of the mean. Throughout this paper, if two values are within 5%, we do not consider that a significant difference.

3.2 PostMark

PostMark focuses on stressing the file system by performing a series of file system operations such as directory lookups, creations, appends, and deletions on small files. A large number of small files is common in electronic mail and news servers where multiple users are randomly modifying small files. We configured PostMark to create 20,000 files and perform 200,000 transactions in 200 subdirectories. To simulate many concurrent users, we ran each configuration with 1, 2, 4, 8, 16, and 32 concurrent PostMark processes. The total number of transactions, initial files, and subdirectories was divided evenly among each process. We chose the above parameters for the number of files and transactions as they are typically used and recommended for file system benchmarks [29, 66]. We used many subdirectories for each process, so that the work could be divided without causing the number of entries in each directory to affect the results (Ext2 uses a linear directory structure). We ran each test at least ten times.

Through this test we demonstrate the overhead of file system operations for each system. First we discuss the results on our workstation configuration in Section 3.2.1. Next, we discuss the results on our file server configuration in Section 3.2.2.

3.2.1 Workstation Results

Ext2 Figure 3.1 shows elapsed time results for Ext2. For a single process, Ext2 ran for 121.5 seconds, used 12.2 seconds of system time, and average CPU utilization was 32.1%. When a second process was added, elapsed time dropped to 37.1 seconds, less than half of the original time. The change in system time was negligible at 12.5 seconds. System time remaining relatively constant is to be expected because the total amount of work remains fixed. The average CPU utilization was 72.1%. For four processes the elapsed time was 17.4 seconds, system time was 9.9 seconds, and CPU utilization was 79.4%. After this point the improvement levels off as the CPU and disk became saturated with requests.

LOOPDEV Figure 3.2(a) shows elapsed time results for LOOPDEV, which follows the same general trend as Ext2. The single process test is dominated by I/O and subsequent tests improve substantially, until the CPU is saturated. A single process took 126.5 seconds for Null, 106.7 seconds for Blowfish, 111.8 seconds for AES, and 136.3 seconds for 3DES. The overhead over Ext2 is 4.2% for Null, -12.2% for Blowfish, -8.0% for AES, and 12.25% for 3DES. The fact that Blowfish and AES are faster than Ext2 is an artifact of the disks we used. When we ran the experiment on a ramdisk, a SCSI disk, or a slower IDE disk, the results were as expected. Ext2 was fastest, followed by Null, Blowfish, AES, and 3DES. The system times, shown in Figure 3.2(b), were 12.4, 13.5, 14.0, and 27.4 seconds for Null, Blowfish, AES and 3DES, respectively. The

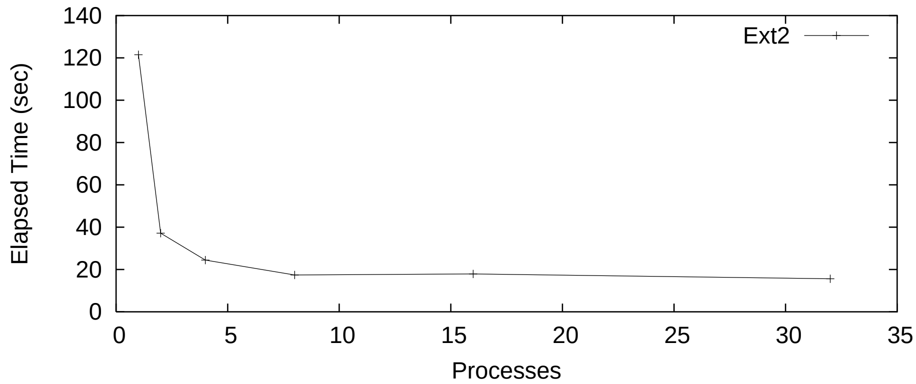
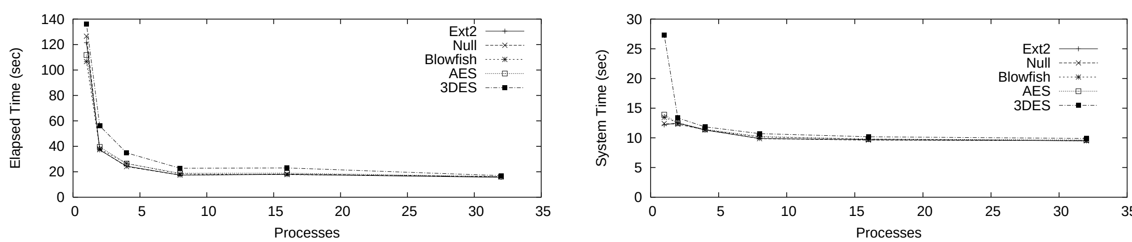


Figure 3.1: PostMark: Ext2 elapsed time for the workstation

average CPU utilization was 31.6%, 38.3%, 36.6%, 40.0% for Null, Blowfish, AES and 3DES, respectively. When there were eight concurrent processes, the elapsed times were 17.3, 18.4, 18.8, and 22.8 seconds, respectively. The overheads were 1.0–32.2% for eight processes. The average CPU utilization ranged from 64.1–80.1%. The decline in system time was unexpected, and is due to the decreased elapsed time. Since dirty buffers have a shorter lifetime when elapsed time decreases, `kupdated` does not flush the short-lived buffers, thereby reducing the amount of system time spent during the test.



(a) Elapsed time

(b) System time

Figure 3.2: PostMark: LOOPDEV *for the workstation*

The results fit the pattern established by Ext2 in that once the CPU becomes saturated, the elapsed time remains constant. Again, the largest benefit is seen when going from one to two processes. Encryption does not have a large user visible impact, even for this intense workload.

LOOPDD Figure 3.3 shows the elapsed time results for the LOOPDD configuration. The elapsed times for a single process are 41.4 seconds for Null, 44.5 for Blowfish, 45.7 for AES, and 73.9 for 3DES. For eight processes, the elapsed times decrease to 23.1, 23.5, 24.5, and 50.39 seconds for Null, Blowfish, AES, and 3DES, respectively.

With LOOPDD we noticed an anomaly at four processes. We have repeated this test over 50 times and the standard deviation remains high at 40% of the mean for Null, AES, and Blowfish. There is also an inversion between the elapsed time and the cipher speed. The Null cipher is the slowest, followed by Blowfish, and then AES. 3DES is not affected by this anomaly. We have investigated this anomaly in several ways. We know this to be an I/O effect, because the system time remains constant and when the test is run with a ram disk, the anomaly disappears. We also ran the test for the surrounding points at three, five, six, and seven

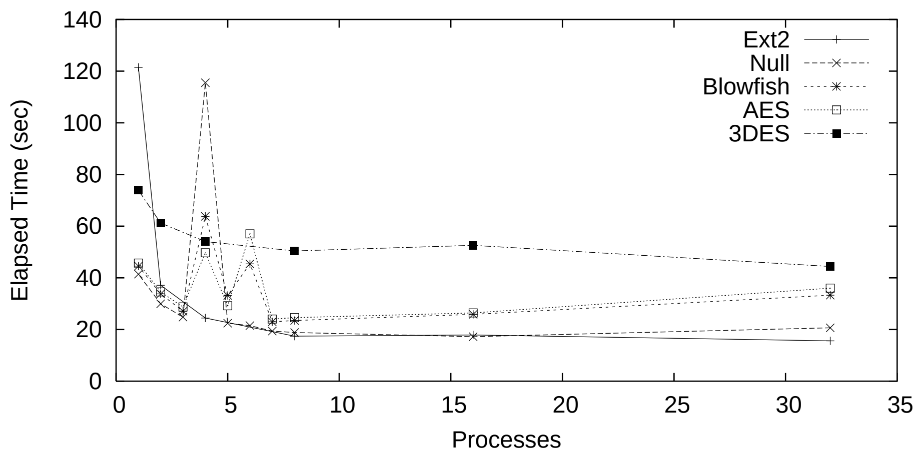


Figure 3.3: Note: Null, Blowfish, and AES additionally have points for 1–8 in one process increments rather than exponentially increasing processes.

concurrent processes. For Null there were no additional anomalies, but for Blowfish and AES the results for 4–6 process were erratic and standard deviations ranged from 25–67%. We have determined the anomaly's cause to be an interaction with the Linux buffer flushing daemon, `bdflush`. Flushing dirty buffers in Linux is controlled by three parameters: $nfract$, $nfract\_stop$, and $nfract\_sync$. Each parameter is a percentage of the total number of buffers available on the system. When a buffer is marked dirty, the system checks if the number of dirty buffers exceeds $nfract\%$ (by default 30%) of the total number of buffers. If so, `bdflush` is woken up and begins to sync buffers until only $nfract\_stop\%$ of the system buffers are dirty (by default 20%). After waking up `bdflush`, the kernel checks if more than $nfract\_sync\%$ of the total buffers are dirty (by default 60%). If so, then the process synchronously flushes `NRSYNC` buffers (hard coded to 32) before returning control to the process. The $nfract\_sync$ is designed to throttle heavy writers and ensure that enough clean buffers are available. We changed the $nfract\_sync$ parameter to 90% and reran this test. Figure 3.4 shows the results when using $nfract\_sync = 90\%$ and the anomaly is gone. Since the Null processes are writing to the disk so quickly, they end up causing the number of dirty buffers to go over the $nfract\_sync$ threshold. The four process test is very close to this threshold, which accounts for the high standard deviation. When more CPU is used, either through more processes or slower ciphers, the rate of writes is slowed, and this this effect is not encountered. We have confirmed our analysis by increasing the RAM on the machine from 512MB to 1024MB, and again the elapsed time anomaly disappeared.

The other major difference between LOOPDD and LOOPDEV is that LOOPDD uses more system time. This is for two reasons. First, LOOPDD traverses the file system code twice, once for the test file system and once for the file system containing the backing store. Second, LOOPDD effectively cuts the buffer and page caches in half by double buffering. Cutting the buffer cache in half means that fewer cached cleartext pages are available so more encryption operations must take place. The system time used for LOOPDD is also relatively constant, regardless of how many processes are used. We instrumented a CryptoAPI cipher to count the number of bytes encrypted, and determined that unlike LOOPDEV, the number of encryptions and decryptions does not significantly change. The LOOPDEV system marks the buffers dirty and issues an I/O request to write that buffer. If another write comes through before the I/O is completed, then the writes are coalesced into a single I/O operation. When LOOPDD writes a buffer it adds the buffer to the end of a queue for the loop thread to write. If the same buffer is written twice, the buffer is added to the queue twice, and hence encrypted twice. This prevents searching through the queue, and since the lower-level file system may in fact coalesce the writes, this does not have a large impact on elapsed time.

The results show that LOOPDD systems have several complex interactions with many components of the

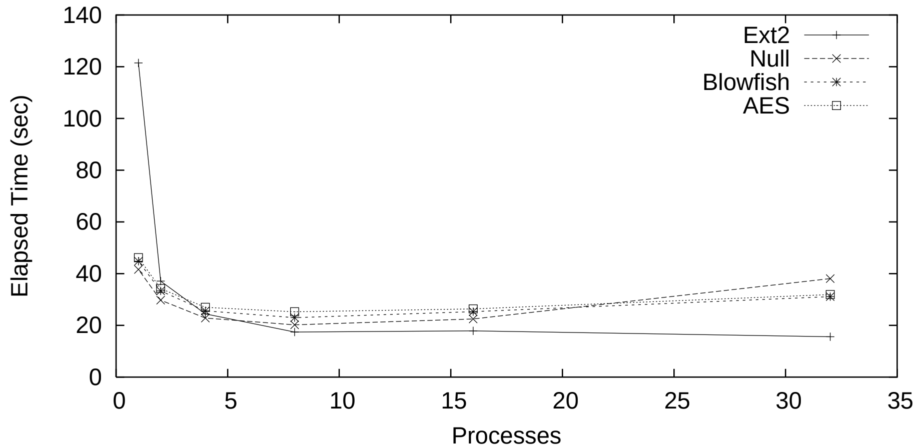


Figure 3.4: Note: 3DES is not included because it does not display this anomaly.

system that are difficult to explain or predict. When maximal performance and predictability is a consideration, LOOPDEV should be used instead of LOOPDD.

CFS Figure 3.5 shows CFS elapsed times, which are relatively constant no matter how many processes are running. Since CFS is a single threaded NFS server, this result was expected. System time also remained constant for each test: 146.7–165.3 seconds for Null, 298.9–309.4 seconds for Blowfish, and 505.7–527.8 seconds for 3DES. There is a decrease in elapsed time when there are eight concurrent processes (27.0% for Null, 14.1% for Blowfish, and 7.8% for 3DES), but the results return to their normal progression for 16 processes. The dip at eight processes occurs because I/O time decreases. At this point the request stream to our particular disk optimally interleaves with CPU usage. System time remains the same. When this test is run inside of a ram disk or on different hardware, this anomaly disappears.

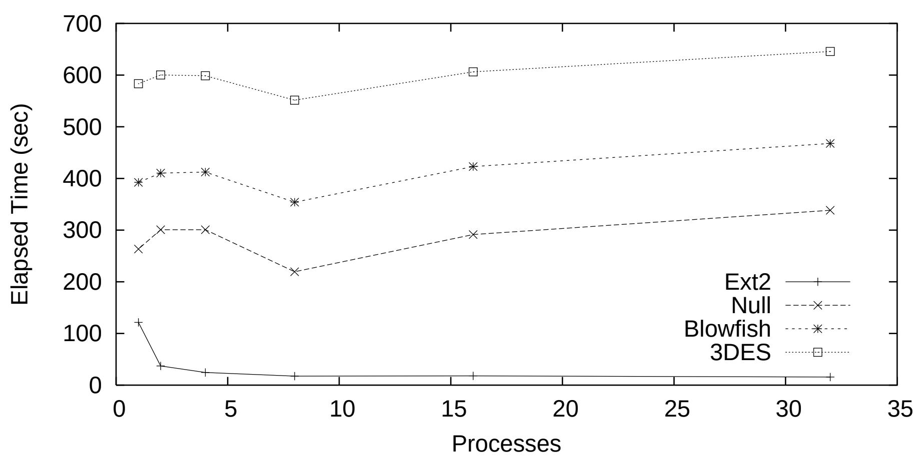
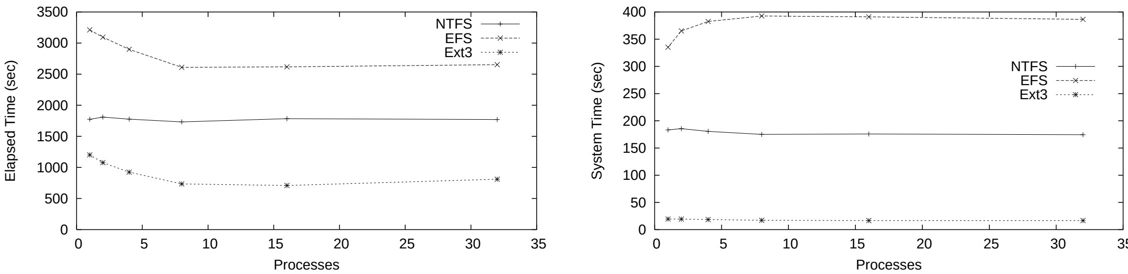


Figure 3.5: PostMark: CFS elapsed time for the workstation

We conclude that both the user-space and single-threaded architecture are bottlenecks for CFS. The single threaded architecture prevents CFS from making use of parallelism, while the user-space architecture causes CFS to consume more system time for data copies to and from user space and through the network stack.



(a) Elapsed time (b) System time

Figure 3.6: PostMark: NTFS on the workstation.

NTFS Figure 3.6(a) shows that NTFS performs significantly worse than Ext2/3 for the PostMark benchmark. The elapsed time for NTFS was relatively constant regardless of the number of processes, ranging from 28.9–30.2 minutes. Katcher reported that NTFS performs poorly for this workload when there are many concurrent files [29]. Unlike Ext2, NTFS is a journaling file system. To compare NTFS with a similar file system we ran this benchmark for Ext3 with journaling enabled for both meta-data and data. Ext2 took 2.1 minutes for one process, and Ext3 took 20.0 minutes. We hypothesize that the journaling behavior of NTFS negatively impacts its performance for PostMark operations. NTFS has many more synchronous writes than Ext3, and all writes have three phases that must be synchronous. First, changes to the meta-data are written to allow a read to check for corruption or crashes. Second, the actual data is written. Third, the metadata is fixed, again with a synchronous write. Between the first and last step, the metadata may not be read or written to, because it is in effect invalid. Furthermore, the on-disk layout of NTFS is more prone to seeking than that of Ext2 (Ext3 has an identical layout). NTFS stores most metadata information in the *master file table* (MFT), which is placed in the first 12% of the partition. Ext2 has group descriptors, block bitmaps, inode bitmaps, and inode tables for each cylinder group. This means that a file's meta-data is physically closer to its data in Ext2 than in NTFS.

When an encrypted folder is used for a single process, elapsed time increases to 57.2 minutes, a 98% overhead over NTFS. The right of Figure 3.6(b) shows that for a single process, system time increases to 335.2 seconds, a 82.9% overhead over NTFS. We have observed that unlike NTFS, EFS is able to benefit from parallelism when multiple processes are used under this load. When there are eight concurrent processes, the elapsed time decreases to 43.5 minutes, and the system time increases to 386.5 seconds. No additional throughput is achieved after eight processes. We conclude that NTFS, and hence EFS, is not suitable for busy workloads with a large number of concurrent files.

NCryptfs Figure 3.7 shows that for a single process running under NCryptfs, the elapsed times were 136.2, 150.0, 156.5, and 368.9 seconds for Null, Blowfish, AES, and 3DES, respectively. System times were 22.2, 44.0, 54.3, and 287.5 seconds for Null, Blowfish, AES, and 3DES, respectively. This is higher than for LOOPDD and LOOPDEV because NCryptfs uses write-through, so all writes cause an encryption operation to take place. The system time is lower than in CFS because CFS must copy data to and from user-space, and through the network stack. Average CPU utilization was 36.4% for Null, 47.5% for Blowfish, 52.2% for AES, and 84.9% for 3DES. For eight processes, elapsed time was 27.0, 48.2, 56.2, and 281.5 seconds for Null, Blowfish, AES, and 3DES, respectively. CPU utilization was 90.2%, 96.3%, 98.0%, and 99.3% for Null, Blowfish, AES, and 3DES, respectively. This high average CPU utilization indicates that CPU is the bottleneck, preventing greater throughput from being achieved.

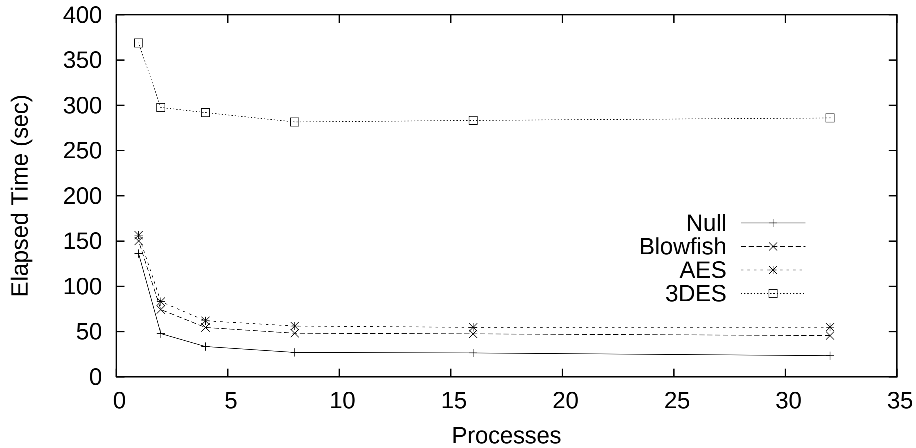


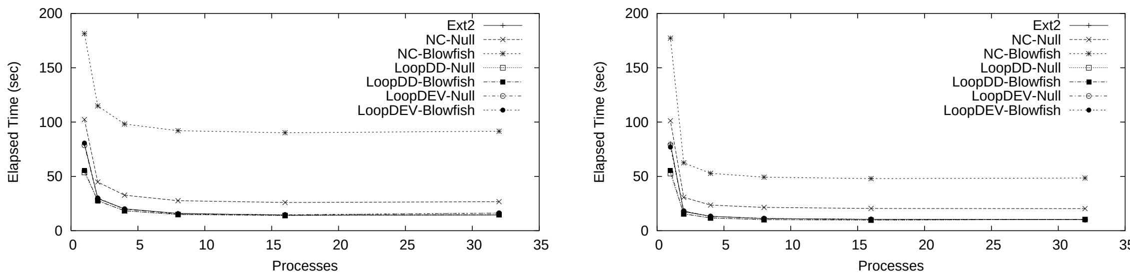
Figure 3.7: PostMark: NCryptfs Workstation elapsed time

Ciphers As expected throughout these tests, 3DES introduces significantly more overhead than AES or Blowfish. Blowfish was designed with speed in mind, and a requirement for AES was that it would be faster than 3DES. In this test AES and Blowfish perform similarly.

We chose to use Blowfish for further benchmarks because it is often used for file system cryptography and has properties that are useful for cryptographic file systems. First, Blowfish supports long key lengths, up to 448-bits. This is useful for long-term storage because it is necessary to protect against future attacks. Second, Blowfish generates tables to speed the actual encryption operation. Key setup is an expensive operation, which takes 4168 bytes of memory and requires 521 iterations of the Blowfish algorithm [57]. Since key setup is done infrequently, this does not place undue burden on the user, but makes it somewhat more difficult to mount a brute-force attack.

3.2.2 File Server Results

Single CPU Itanium The left of Figure 3.8(a) shows the results when we ran the PostMark benchmark on our Itanium file server configuration with one CPU enabled (we used the same SMP kernel as we did with two CPUS). The trend is similar to that seen for the workstation. For Ext2, the elapsed time for a single process was 79.9 seconds, and the system time was 11.7 seconds. For LOOPDEV, the elapsed time is within 1% of Ext2 for all tests while for LOOPDD the elapsed times decreased by 31% for all tests compared to Ext2. For NCryptfs the elapsed time was 102.4 seconds for Null and 181.5 seconds for Blowfish. This entails a 24% overhead for Null and 127% overhead for Blowfish over Ext2. Of note is that NCryptfs uses significantly more CPU time than on the i386. The increase in CPU time is caused by the combination of two factors. First, the Blowfish implementation we used (from OpenSSL 0.9.7b) only encrypts at 31.5MB/s on this machine as opposed to 52.6MB/s on the i386. Second, the i386 architecture uses a 4KB page size by default, whereas the Itanium architecture supports page sizes from 4KB–4GB. The Linux kernel uses a 16KB page size by default. Since NCryptfs encodes data in units of one page, any time a page is updated, NCryptfs must re-encrypt more data. When using a Linux kernel recompiled with a 4KB page size on the Itanium, the system time used by NCryptfs with Blowfish drops by 37.5% to 55.6 seconds. With a 4KB page size, the system time that Blowfish uses over Null on the Itanium is roughly the additional time that the i386 uses, scaled by the encryption speed. Since this test is mostly contained in main memory, NCryptfs also performs significantly more work than does the loopback device in this test. NCryptfs uses write-through to the lower-level file system, so each write operation results in an encryption. The loopback device does not encrypt or decrypt data for operations that are served out of the cache.



(a) One CPU

(b) Two CPUs

Figure 3.8: PostMark: File server elapsed time.

We conclude that the general trends for the Itanium architecture are the same as for the i386 architecture. The i386 has had years of investment in optimizing compilers and optimized code, whereas the Itanium being a new architecture has not yet developed an optimized codebase.

Dual CPU Itanium The right of Figure 3.8(b) shows the results when a second CPU is added to the file server. As expected, the elapsed times for a single process remained unchanged. Again, the bottleneck was the CPU in this experiment, and the additional CPU resources proved to be of benefit. As the number of processes increases past one process, the system time increased slightly for all systems because of the added overhead of locking and synchronization of shared directories and other file-system data structures. However, because the increased CPU time was spread across the two CPUs, elapsed time decreased significantly. The decrease in the elapsed times for Ext2 ranged from 27–30% for eight or more concurrently running processes. LOOPDD had similar behavior for both Blowfish and Null ciphers. For LOOPDEV the decrease ranged from 28–37% for both Blowfish and Null ciphers. The elapsed times for NCryptfs changed to 53% of their value (compared to a single CPU) for Blowfish while the decrease for the Null cipher was 23%.

We conclude that these file systems scale as well as Ext2, and when encryption is added, the benefit is even more significant.

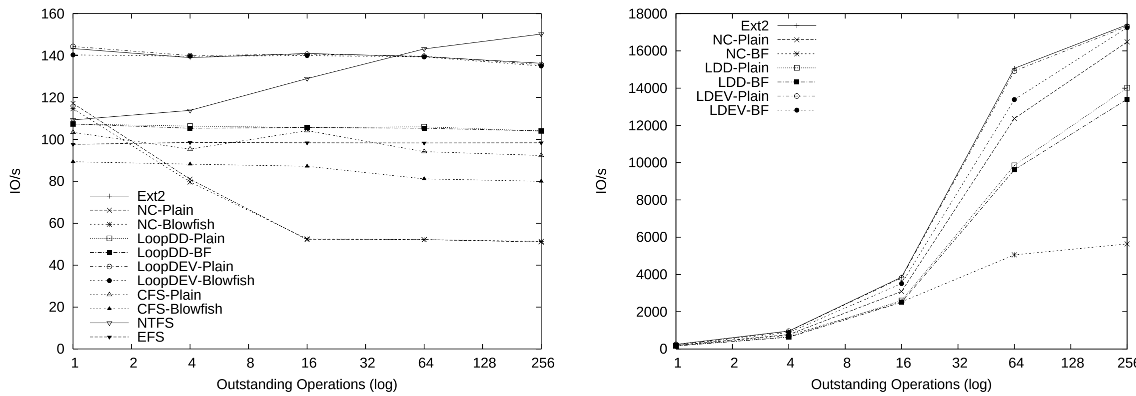
3.3 PGMeter

PenguinoMeter is a file-I/O benchmark with a workload specification patterned after that of the IOMeter benchmark available from Intel [1, 7]. PGMeter measures the data transfer rate to and from a single file. We ran PGMeter with the `fileserver.icf` workload distributed with PGMeter and IOMeter, which is a mixture of 512B–64KB transfers of which 80% are read requests and 20% are write requests. Each operation begins at a randomly selected point in the file. We ran this workload for 1–256 outstanding I/O requests, increasing the number by a factor of 4 on each subsequent run.

Each test starts with a 30 second ramp-up period in order to avoid transient startup effects, and then performs operations for 10 minutes. Every 30 seconds during the test, PGMeter reports the number of I/O operations performed and the number of megabytes of data transferred per second. We ran each test three times, and observed stable results both across and within tests. Since the amount of data transferred is a multiple of the I/O operations, we do not include this result.

For the workstation configuration the data file size was 1GB, and for the file server it was 16GB. This is twice the amount of memory on the machine, in order to measure I/O performance and not performance when reading or writing to the cache.

On Windows we used version 1.3.22-1 of the Cygwin environment to build and run PGMeter. When given the same specification, PGMeter produces similar workloads as IOMeter [7]. A major limitation with PGMeter and Windows is that Cygwin does not support more than 63 concurrent threads, so a large number of outstanding operations can not be generated. We used IOMeter on Windows with the same workload as PGMeter on Linux. This allowed to compare NTFS and EFS for an increasing number of concurrent operations. Though IOMeter and PGMeter are quite similar, conclusions should not be drawn between NTFS and EFS vs. their Linux counterparts.



(a) Workstation

(b) File server

Figure 3.9: The workstation is on the left, and the file server is on the right.

Figure 3.9(a) shows the operations per second (ops/sec) for PGMeter and IOMeter. As in previous tests we adopted Ext2 as the baseline for the Linux systems. The average I/O ops/sec for Ext2 started at 143 ops/sec for one outstanding operation (each outstanding operation uses a thread on Linux) and dropped to 136.3 ops/sec for 256 threads. The ops/sec for LOOPDEV with Null ranged from 144 ops/sec for one thread to 135.7 ops/sec for 256 threads. When using Blowfish, ops/sec ranged from 140–135 ops/sec. LOOPDD dropped to 107 ops/sec for one thread and 104 ops/sec for 256 threads. This is because LOOPDD must pass through the file system hierarchy twice. The results for Blowfish were identical, indicating that the encryption overhead is not significant. IOMeter with the file server workload under NTFS showed a performance improvement when the number of outstanding operations was increased. For one outstanding operation, 83.7 ops/sec were performed, increasing to 100 ops/sec for 256 outstanding operations. EFS followed suite increasing from 78.7 to 91.7 ops/sec. From this we conclude that even though NTFS performs poorly with many concurrent files, it is able to effectively interleave CPU and disk resources when there is only one open file.

As on the other Linux-based systems, CFS showed the same pattern of a decrease in the ops/sec as the number of threads increased. The range for Null was 107.6 ops/sec for one thread to 92.4 ops/sec for 256 threads. For CFS with Blowfish, the number of ops/sec ranged from 89.3 for one thread to 80 for 256 threads. NCryptfs performed better than LOOPDD and CFS for a single thread, achieving 117 ops/per second for Null and 114 for Blowfish. However, for multiple threads, NCryptfs with Null quickly dropped to 81 ops/sec for 4 threads and 52.2 ops/sec for 16 threads where it began to stabilize. NCryptfs with Blowfish followed the same pattern, stabilizing at 52.1 ops/sec for 16 threads. The reason that NCryptfs performed poorly for multiple threads is that it must down the inode semaphore for each write operation, so they become serialized and there is significant contention. This behavior was not exhibited during PostMark since each process worked on a disjoint set of files.

Figure 3.9(b) shows the ops/sec achieved on our Itanium fileserver machine. On this machine, we observed an improvement in the ops/sec with an increase in the number of threads due to the interleaving of CPU bound and I/O bound tasks. The number of ops/sec for Ext2 ranged from 244 for one thread to 17,404 for 256 threads. LOOPDEV with Null closely followed Ext2 with 242–17,314 ops/sec. LOOPDEV with Blowfish started off at 220 ops/sec for one thread and reached 17,247 ops/sec for 256 threads. LOOPDD performed fewer ops/sec for both Null and Blowfish, because it had to travel through the file system stack twice. Additionally, LOOPDD cuts the buffer cache in half so fewer requests are served from memory. LOOPDD with Null performed 166–14,021 ops/sec. With Blowfish, the number of ops/sec ranged from 160–13,408. NCryptfs with Null started at 193.3 ops/sec for one thread and increased to 16,482 ops/sec for 256 threads. When using Blowfish, NCryptfs only achieved 176–5,638 ops/sec, due to the computation overhead of Blowfish combined with the write-through implementation, and the larger 16KB default (PAGE_SIZE) unit of encryption.

From this benchmark we conclude that I/O intensive workloads suffer a greater penalty than meta-data intensive workloads on cryptographic file systems. There are several reasons for this: double buffering cuts page caches in half, data must be copied through multiple file systems, and there is increased lock contention. Additionally, the increased unit of encryption begins to have a negative effect on the systems. Though Blaze's hybrid OFB/ECB approach does not suffer from increasing page or block sizes, its security not well studied.

3.4 AIM Suite VII

The AIM Multiuser Benchmark Suite VII (AIM7) is a multifaceted system-level benchmark that gives an estimate of the overall system throughput under an ever-increasing workload. AIM7 comes with preconfigured standard workload mixes for multiuser environments. An AIM7 run comprises of a series of sub-runs with the number of tasks increasing in each sub-run. AIM7 creates one process for each simulated operation load (task). Each sub-run waits for all children to complete their share of operations. We can specify the number of tasks to start with and the increment for each subsequent subrun. The AIM7 run stops after it has reached the *crossover* point that indicates the multitasking operation load where the system's performance becomes unacceptable, i.e., less than 1 job/minute/task. Each run takes over 12 hours and is self-stabilizing, so we report the values from only a single run. In our other trials have observed the results to be stable over multiple runs. AIM7 runs a total of 100 tests for each task, selecting the tests based on weights defined by the workload configuration. We chose the fileserver configuration that represents an environment with a heavy concentration of file system operations and integer computations. The file server configuration consists of 40% asynchronous file operations (directory lookup, random read, copy, random writes, and sequential writes), 58% miscelleneous tests that do not depend on the underlying file system and 2.5% synchronous file operations (random write, sequential write and copy). AIM7 reports the total jobs/minute and jobs/minute/task for each subrun. For a typical system, as the number of concurrently running tasks increases, the jobs/minute increases to a peak value as idle CPU time is used; thereafter it reduces due to software and hardware limitations. We ran AIM7 with one initial task and an increment of two tasks for each subrun. AIM7 uses an *adaptive timer* to determine the increment after the first eight subruns. After the invocation of the adaptive timer the number of tasks depends on the recent rate of change of throughput, but the adaptive timer often significantly overshoots the crossover point [2]. AIM7 can also use Newton's method to approach the crossover point more gradually. We opted to use Newton's method.

The results of AIM7 can be seen in Figure 3.10. Ext2 served as a base line for all the tests. The peak jobs/minute on Ext2 was 230 with 5 tasks and the crossover point was 112 tasks. LOOPDEV with a Null cipher peaked at 229 jobs/minute with 5 tasks and reached crossover at 108 tasks. With Blowfish, the peak jobs/minute was 217 with 5 tasks and the crossover was at 114 tasks. LOOPDD, NCryptfs, and CFS all achieved a higher peak jobs/minute and crossover point than Ext2. This is because they do not respect

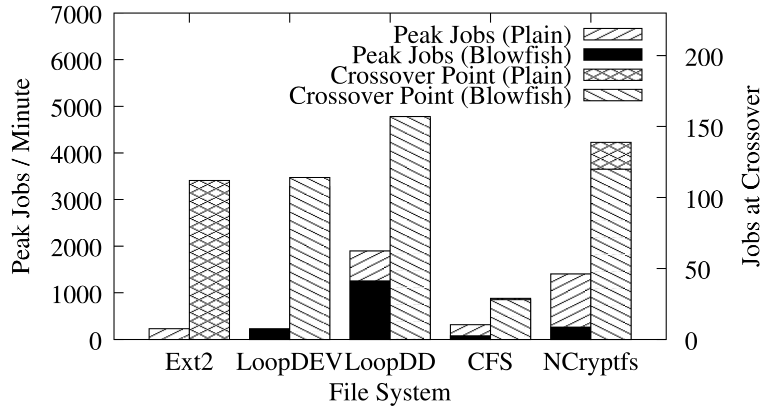


Figure 3.10: Note: Peak jobs/minute is using the left bars and scale. Crossover is using the right bars and scale.

synchronous write requests. When run with only asynchronous writes, Ext2 peaks at 2253 jobs/minute. LOOPDD peaked at 1897 jobs/minute with 46 tasks using Null; and 1255 jobs/minute with 40 tasks using Blowfish. The crossover point was 156 tasks for Null, and 157 tasks for Blowfish. This difference is not significant. NCryptfs with Null peaked at 1405 jobs/minute with 34 tasks, and reached the crossover at 139 tasks. With Blowfish, NCryptfs peaked at 263 jobs/minute with 23 tasks, and reached the crossover at 120 tasks. When using Blowfish with NCryptfs, the write-through behavior causes CPU utilization to increases and the integer computation portion of the AIM7 test is not able to achieve as high a throughput. CFS with Null peaked at only 317 jobs/minute with 11 tasks, despite the fact that it does not respect the synchronous flag. This is because there is a significant amount of overhead incurred by traversing the network stack and copying data to and from user space. The crossover point was 29 tasks. When CFS is used with Blowfish, the peak throughput is reduced to 77 jobs/minute with 9 tasks. The crossover is reached at 28 tasks.

From these results we conclude that synchronous I/O may be a small part of most workloads, but it has an overwhelming effect on the total throughput. We also conclude that as workloads shift to an intensive mix of I/O and CPU usage, encryption begins to affect throughput more throughput. Our previous experience shows that less intense workloads, such as compiles, have little user visible overhead [69].

On the Itanium file servers, Ext2 reached a peak jobs/sec count of 188 for 93 tasks and reached crossover at 171 tasks. LOOPDEV again closely mimicked the behavior of Ext2 with both Null and Blowfish reaching a peak of 171 jobs/minute with 103 tasks for Null and 117 tasks for Blowfish. The crossover points were 166 and 165 for Null and Blowfish, respectively. If only the asynchronous I/O portions are executed, Ext2 has a peak throughput of 4524. As compilers for the Itanium improve, the computational portions of this suite of tests should also improve to exceed the throughput of the i386.

LOOPDD with Null and Blowfish showed the same trend as the workstation machines. With Null, the peak jobs/min was 3315 with 48 tasks, and the crossover point 934 tasks. Blowfish peaked at 1613 jobs/min with 200 tasks, and the crossover point was 1049. Since the Blowfish implementation is slower on the Itanium, the peak jobs/min dropped by 51% when compared with Null. On the i386 the peak jobs/min only dropped by 34%. NCryptfs with Null peaked at 2365 jobs/min with 167 tasks and the crossover point was 1114 tasks. With Blowfish, NCryptfs becomes significantly slower, peaking at 132 jobs/min with 132 tasks and reaching crossover at 133 tasks. This can again be attributed to the interference with CPU-bound processes, and is only exacerbated by compilers not yet optimizing the Blowfish implementation as well as on the i386.

From this we conclude that even though the raw I/O performance is significantly better on the Itanium, a mix of heavily compute and I/O intensive workloads still suffer from the slower CPU. When using a cryptographic file system with write-through this is only worsened. We expect that as Itanium compilers improve, this problem will be less pronounced.

3.5 Other Benchmarks

In this section we explain interesting results involving configurations not previously discussed.

Key Size A common assumption is that stronger security decreases performance. A classic example of this is that if you use longer key lengths, then encryption is more costly. We set out to quantify exactly how much it would cost to increase your key length to the maximum allowed by our ciphers. We used two variable key length ciphers: Blowfish and AES.

Cipher	Key Size (bits)	Speed MB/s
AES	128	27.5
AES	192	24.4
AES	256	22.0
Blowfish	Any	52.6
3DES	168	10.4
Null	None	694.7

Table 3.2: Workstation raw encryption speeds for AES, Blowfish and 3DES.

To encrypt data, most block ciphers repeat a simple function several times; each time the primitive function is iterated is called a *round*. Blowfish has 16 rounds no matter what key size you choose; only the key expansion changes, which occurs only on initialization. This means that encryption time remains constant without consideration to whether a 32 or 448 bit key is chosen. Our experimental results confirmed this. No significant difference was recorded for raw encryption speeds for NCryptfs running the PostMark benchmark. On the other hand, AES has 10, 12, or 14 rounds depending on the key size (128, 192, and 256 bits, respectively). Table 3.2 shows raw encryption speeds on our workstation configuration. There is a 24.7% difference between the fastest and slowest AES key length, but when benchmarked in the context of file systems, this translates into a mere 1.3% difference for NCryptfs running PostMark. This difference is within the margin of error for these tests (here, standard deviations were 1–2% of the mean).

The overhead of encryption is largely masked by I/O and other components of the system time, such as the VFS code. Since the key size plays a small role in the file system throughput, it is wise to use as large a key size as possible. This is made only more important by the persistent nature of file systems, where there is often lots of data available to a cryptanalyst — some of which may be predictable, e.g., partition tables [55]. Since file systems are designed for long-term storage, it is important that data is protected not only against adversaries of today, but also of the future.

Sparse Loopback Devices Throughout our previous tests, LOOPDD used a preallocated file that filled the test partition to reduce I/O effects. It is often the case that an encrypted volume is only a small fraction of the disk. This leads to interesting results with respect to I/O. Ext2 attempts to put blocks from the same file into a single cylinder group; and spread directories evenly across cylinder groups with the goal of clustering related files [42].

Using a preallocated file as large as the partition does not drastically affect the Ext2 allocation policy. However, if the preallocated file is smaller than the partition, it is clustered into as few cylinder groups as possible, thereby reducing the amount of seeking required. Using a sparse file as the backing store results in an even more optimal allocation policy than the small file, since holes in the upper-level file system are not allocated.

We ran this test on our workstation with a 10GB Seagate 5400 RPM IDE hard drive. We used a slower disk than in previous tests to accentuate the differences between these configurations. Using the entire drive formatted with Ext2, PostMark took 551 seconds and used 12.2 seconds of system time. When using a 500MB LOOPDD file, Null mode took 79.6 seconds, and used 27.1 seconds of system time. Using Cryptoloop with a sparse backing store took 51.2 seconds and used 27.2 seconds of system time.

From this test we conclude that although using a loop device increases system overhead, the effects that it has on layout policy can be beneficial. On the other hand, the fact that LOOPDD can so markedly affect the layout policies of the file system may have negative impacts for some workloads. One advantage of systems like LOOPDEV is that they will not affect the layout decisions made by the upper level file system.

Chapter 4

Snapshotting and Versioning Systems Survey

In this section we examine *snapshotting* and *versioning*. We define snapshotting as the act of creating a consistent view of an entire file system, and versioning as making historical backup copies of individual files. Like cryptographic file systems, snapshotting and versioning systems can operate at several levels: at the block-level, as a disk-based file system, as a stackable file system, as a user-level library, or at the application level.

4.1 Block-Based Systems

4.1.1 FreeBSD Soft Updates

FreeBSD Soft Updates use snapshots to ensure file system consistency [36]. In FreeBSD FFS, a technique called Soft Updates is used to ensure that on-disk data structures are always consistent. Soft Updates track pointer dependencies within the file system, and ensure that the only possible on-disk inconsistency is orphaned resources (i.e., resources that are allocated, but have no pointer to them). Soft Updates allow the machine to restart without the need to immediately run `fsck`. However, it is useful to run `fsck` in the background to reclaim the orphaned blocks and inodes. `fsck` takes a snapshot, performs checks for orphaned resources, and then informs the OS of the orphaned resources via a new system call, which reclaims the orphaned resources.

In FreeBSD, snapshots are managed by the file system, but logically occur at the block-level. To take a snapshot, the file system writes all pending data, and creates a new snapshot inode. Each time a disk-block is updated, the file system checks if that block is not already in the snapshot and performs copy-on-write if the snapshot does not already have a copy. To view a snapshot, it is mounted like any other file system (using the *vnd* driver so that the snapshot file appears as a device). Snapshots are possible using Soft Updates, because the on-disk structure is always consistent. Without Soft Updates, the status of the file system at any point in time may not be consistent and errors would occur.

4.1.2 Clotho

Clotho is a block-based snapshotting system that is fully file-system agnostic [16]. Clotho stacks on top of a device, and logically divides the device into a *primary data segment* (PDS) and *backup data segment* (BDS). Only the space in the PDS is exposed to the higher-level block device. If a block is written to, then if the block is only part of the current version, the existing block is simply overwritten. If the block is part of a past version, then the data is written in a new location on the physical device. By writing to a new location, Clotho does not use I/O bandwidth to make a copy of the old data. This allows Clotho to be used for any file system (e.g., Ext2 or Reiserfs); or even for database servers that require a raw partition. Clotho uses block differences to compact snapshots to reduce space utilization within the BDS.

4.1.3 Venti

Venti is not a snapshotting system, but rather an archival block-based storage system that a snapshotting system could be based on. In Venti the blocks are addressed by a unique hash of the block's contents [50]. Since the block's contents and address are interchangeable, Venti is inherently a write-once system. Since a block is addressed by its contents, there are no duplicate blocks in the file system. This property can be exploited for building an efficient snapshotting system, because space is not utilized for the duplicated data within snapshots.

4.2 Disk-Based File Systems

4.2.1 Write Anywhere File Layout

The Write Anywhere File Layout (WAFL) was designed by Network Appliance specifically for NFS servers [24]. WAFL uses inodes like FFS-like file systems, but stores meta-data within files. One important characteristic of WAFL is the ability to take a read-only snapshot of the file system. Users can access old versions through a `.snapshot` directory in each directory. Since WAFL uses files instead of fixed locations (e.g., at the beginning of cylinder groups as in FFS-like file systems) to store meta-data, updated meta-data can be located anywhere on disk. In WAFL, the root inode describes the inode meta-data file. To create a snapshot, the root inode is copied. On subsequent writes, copy-on-write is used to create the backup copies. When WAFL uses copy-on-write to avoid duplicating disk blocks that are common between a snapshot and the active file system.

Episode [9] is similar to WAFL but less efficient. Episode must copy the entire inode tree for each snapshot, whereas WAFL must copy only the root inode.

4.2.2 Ext3cow

Ext3cow extends Ext3 to support snapshots by changing ext3's on-disk and in-memory meta-data structures [44]. To take a snapshot, ext3cow adds the current time to a list of snapshots in the super-block. Each inode is augmented with an epoch. When a write to an inode crosses a snapshot boundary, per-block copy-on-write takes place. One interesting aspect of Ext3cow is that it avoids data copies within memory and does not duplicate pages or buffers within the cache. To access a snapshot, a user opens *`file@epoch`*, where *file* is the filename and *epoch* is the number of seconds since the UNIX epoch (January 1, 1970).

4.2.3 Elephant File System

The Elephant file system is a copy-on-write versioning file system implemented on the FreeBSD 2.2.8 kernel [56]. Elephant transparently creates a new version of a file on the first write to an open file. Elephant also provides users with four retention policies: keep one, keep all, keep safe, and keep landmark. Keep one is no versioning; keep all retains every version of a file; keep safe keeps versions for a specific period of time but does not retain the long term history of the file; and keep *landmark* retains only important versions in a file's history. A user can mark a version as a landmark or the system uses heuristics to mark other versions as landmark versions. Elephant also provides users with the ability to register their own space reclamation policies.

4.2.4 Comprehensive Versioning File System

The Comprehensive Versioning File System (CVFS) is a versioning system designed with security in mind [62, 64]. Each individual write or the smallest meta-data change (including atime updates) results in a new version. Since many versions are created, new on-disk data structures were designed so that old versions can be stored and accessed efficiently. As CVFS was designed for security purposes, it does not have facilities for the user to access or customize versioning.

4.3 Stackable File Systems

4.3.1 Unionfs Snapshotting and Sandboxing

Fan-out file systems are a new class of stackable file systems that layer functionality on top of several existing file systems. As can be seen in Figure 4.1, a fan-out file system, like Unionfs, calls several underlying file systems in turn [20]. Unionfs virtually merges or unifies several underlying directories into a single view. It is difficult to provide a perfect illusion of a virtual unification of a set of files while maintaining Unix semantics. Since Unionfs calls several underlying file systems, each of which returns success or failure, Unionfs must carefully order operations so that the user-visible file system reflects the state of the whole union. To preserve consistency, Unionfs defines a precedence for branches, and carefully orders operations to ensure that the user is presented with a consistent view.

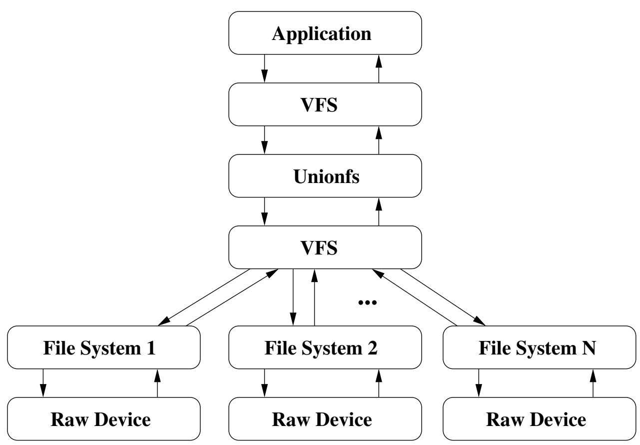


Figure 4.1: Call paths of fanout file systems.

Unionfs is the first implementation of a true stackable fan-out file system. All underlying branches are directly accessed by Unionfs which allows it to be more intelligent and efficient. Unionfs supports a mix of read-only and read-write branches, features not previously supported on any unioning file system. Unionfs also supports the dynamic addition and deletion of any branch of any precedence, whereas previous systems only allowed the highest or lowest precedence branch to be added or removed.

In FreeBSD, union mounts provide similar functionality, but instead of using a fan-out structure use a linear stack [43]. FreeBSD Union mounts do not support many of the advanced features that Unionfs does, including dynamic insertion and deletion of branches; multiple read-write branches; and support for any underlying file system.

Since Unionfs was designed with versatility in mind, it is useful for new applications. Of particular interest to this paper is Unionfs's ability to take a snapshot. Unionfs defines a priority for branches, branches to the left have a higher priority than branches to the right. For snapshots, branches to the left contain more recent data and branches to the right contain less recent data. To take a snapshot at time t_n, a new high-priority branch (n) is added, and all the lower priority branches ($n-1, n-2, \ldots, 0$) are marked read only. The new branch is originally empty, but once new data is written, it is written to the new branch. To view a snapshot taken at time i, all that needs to be done is unify branches $i-1, i-2, \ldots, 0$. Unionfs transparently migrates files open for writing to the new branch.

Unionfs also supports sandboxing using *split-view* caches. If an intrusion detection system (IDS) detects that a process is suspicious, then the two most common options are to alert the administrator or terminate the process. Sandboxing provides a middle ground: the suspect (or *bad*) process or processes continues to run, but it does not change the data seen by other processes (*good processes*) or have access to newly created data.

4.3.2 Versionfs

In our own recent work we have designed a lightweight user-oriented versioning file system called *Versionfs* [38]. Many writes end up overwriting the same data (e.g., a text editor often rewrites the whole file even if only a few bytes have changed). Versionfs employs a *copy-on-change* technique. After a write occurs, Versionfs compares the old contents with the new contents. If a change was made, then Versionfs creates a new version.

Since Versionfs is stackable, it works with any file system, whether local or remote. Versionfs also provides a host of user-configurable policies: versioning by users, groups, processes, or file names and extensions; version retention policies by maximum number of versions kept, age, or total space consumed by versions of a file; version storage policies using full copies, compressed copies, or page deltas. Versionfs creates file versions automatically, transparently, and in a file-system portable manner—while maintaining Unix semantics.

4.4 User-level Libraries

4.4.1 Cedar File System

The Cedar File System is a distributed file system that supports only immutable files [18]. By supporting only immutable files, Cedar eliminates cache consistency issues on the clients (because the clients will always have a consistent view of each file). In the Cedar File System, users have to copy files to a local disk and edit it there. To version it they have to transfer the file to a remote server (via FTP) and this would create a new immutable version on the server. Cedar is also not portable as it works only with a particular file system.

4.4.2 3D File System

The 3D File System (3DFS) adds a third dimension to the file system [17, 32]. Standard UNIX file systems have two dimensions: the present working directory "`.`" and the parent directory "`..`". 3DFS allows users to change to the "`...`" directory, which exposes previous versions of the file system.

3DFS provides a library that can be used for versioning. Version Control System (VCS) is a prototype that is built to demonstrate the usefulness of the library. Versioning in VCS has to be done with explicit commands like `checkin`, `checkout`, etc. A version file is conceptually organized like a directory, but it returns a reference to only one of the files in the directory. To decide which version must be selected, each process keeps a version-mapping table that specifies the rules of version selection of a file. However, to use this, all programs have to be linked with libraries provided. Another disadvantage of this system is that directories are not versioned.

4.5 Applications

CVS [4] is a popular user-land tool that is used for source code management. It is also not transparent as users have to execute commands to create and access versions. Rational ClearCase [25] is another user-land tool that is used for source code management. However, it requires an administrator dedicated to it and is also expensive. Other notable user-space versioning systems include Subversion [10], OpenCM [59], and BitKeeper [5]. SubVersion is meant to be a replacement for CVS and includes atomic commits, versions are created per commit (as opposed to CVS which independently creates versions for each file that is part of a commit), improved network resource utilization, and efficient handling of binary files. OpenCM is another CVS replacement. OpenCM includes rename support, access controls, cryptographic authentication, and provides integrity checks. BitKeeper is versioning control system centered around source code, and is used for Linux kernel development. BitKeeper has merging algorithms that allow engineers to quickly merge several branches.

The primary disadvantage of using applications solutions is the same as for encryption systems: the user must be aware of the versioning. When versioning is used to avoid data loss, users must proactively commit their data. When versioning for security auditing, these systems are even less useful, because the attacker will not correctly version files (and may even attempt to delete the versions).

Chapter 5

Operating System Changes

In this chapter we discuss various operating system changes we have made to improve extensible secure file systems. In Section 5.1 we describe split-view caches. In Section 5.2 we describe cache-cleaning within NCryptfs. In Section 5.3 we describe file revalidation in the VFS. In Section 5.4 we discuss our implementation of task-private data and on-exit callbacks. In Section 5.5 we describe user-land callbacks from kernel-space. In Section 5.6 we discuss how we implemented ad-hoc groups in NCryptfs.

5.1 Split-View Caches

Normally, the operating system maintains a single view of the namespace for all users. This limits file system functionality. For example, in file cloaking each user can only see the files that he or she has permission to access [63]. This improves privacy and prevents users from learning information about files they are not entitled to access. To implement this functionality in a range-mapping NFS server, caches had to be bypassed. Unionfs can divert any process to an alternative view of the file system. This functionality can be integrated with an IDS to create a sandboxing file system. Using a filter provided by the IDS, Unionfs can direct good processes to one view of the union, and bad processes to another view of the union.

In Linux, each mount point has an associated `vfsmount` structure. This structure points to the super-block that is mounted and its root dentry. It is possible for multiple `vfsmount`'s to point to a single super-block, but each `vfsmount` points to only one super block and root. When the VFS is performing a lookup operation and comes across a mount point, there is an associated `vfsmount` structure. The VFS simply dereferences the root dentry pointer, and follows it into the mounted file system.

To implement split-view caches we modified the generic `super_operations` operations vector to include a new method, *select_super*. Now, when the VFS comes across a mount point, it invokes `select_super` (if it is defined), which returns the appropriate root entry to use for this operation. This rich new interface was accomplished with minimal VFS changes: only eight new lines of core kernel code were required.

Internally, Unionfs needs to support the ability to have multiple root dentries at once. To do this we create a parallel Unionfs view that is almost a completely independent file system. The new view has its own super-block, dentries, and inodes. This creates a parallel cache that is used for each of the views. However Unionfs uses the lower-level file systems' data cache, so the actual data pages are not duplicated. This improves performance and eliminates data cache coherency problems. The two views are connected through the super-blocks so that when the original Unionfs view is unmounted, so are the new views.

Figure 5.1 shows the VFS structures after a process is sandboxed. In this example we started with a union that had two branches. The first branch $t1$ contains the basis, and then $t2$ contains a incremental changes for a snapshot. In the figure the super-block of this union is represented as S_U and its root dentry is D_U. The super-blocks of the lower-level file systems are S_{t1} and S_{t2}, and their root dentries are D_{t1} and D_{t2}. Assume that an IDS detected a suspicious (or bad) process is detected at time $t3$, then it can be sandboxed as follows. First a new view of the union is created for the suspect process or processes. This is accomplished by creating a new super-block and root dentry structure ($S_{U'}$ and $D_{U'}$, respectively). In this view a new branch $b3$ is added as the highest priority branch, and branches $t1$ and $t2$ are marked read-only. In the original view, a new branch $g3$ is added as the highest priority branch, and the existing branches are marked read-only. If the bad process writes data, then the write is directed to b_n and the data seen by good processes is untouched. If a good process writes data, then the bad processes will not see it.

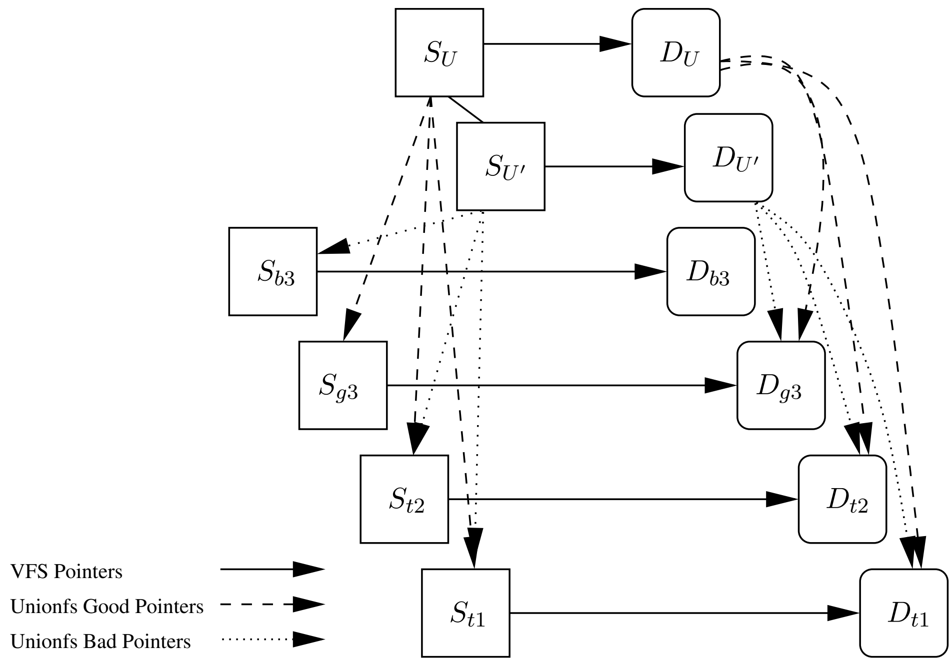


Figure 5.1: S represents a super-block; D represents a root dentry. U is the original Unionfs super-block; U' is alternate or bad view; $b3$ is the branch that is used for sandboxing bad processes; $g3$ is used by good processes; $t2$ and $t1$ form the basis of the snapshot.

5.2 Cache Cleaning

Cleartext pages normally exist in the page cache. Unused name data and metadata may remain in the dcache and icache, respectively. If a system is compromised, then this data is vulnerable to attack. For example, an attacker can examine memory (through `/dev/kmem` or by loading a module). To limit this exposure, NCryptfs evicts cleartext pages from the page cache, periodically and on detach (when the association between the NCryptfs mount and the underlying file system is removed). Unused dentries and inodes are also evicted from the dcache and icache, respectively. For added security at the expense of performance, NCryptfs can purge cleartext data from caches more often. In this situation, the decryption expense is incurred for each file system operation, but I/O time is not increased if the ciphertext pages (e.g., Ext2 pages) are already in the cache. Zero-Interaction Authentication (ZIA), another encryption file system based on Cryptfs, takes the approach of encrypting all pages when an authentication expires [11]. This is less efficient than the NCryptfs method, because ZIA will also maintain copies of pages in the lower-level file system. ZIA requires the initial encryption of pages, and will use memory for these encrypted pages.

5.3 File Revalidation

The VFS provides methods for ensuring that both cached dentry and inode objects are valid before it uses them. To check the freshness of objects, the VFS calls `revalidate` and `d_revalidate` on inodes and dentries, respectively. This is used for NFS, SMBFS, and other network file systems to ensure that the cache did not become stale compared to the server. In Unionfs, the same mechanism is used to efficiently add and remove branches. Unionfs compares the generation numbers in the inode and the dentry with the generation number in the super-block. If the generation numbers do not match, Unionfs refreshes the data structures from the lower-level file systems.

However, file objects have no such revalidation method. This is shortcoming that is especially acute for stackable file systems because the upper-level file object is very much like a cache of the lower-level file object or objects. In Unionfs this becomes important when a snapshot is taken. If the file is not revalidated, then writes can continue to affect read-only branches. With file revalidation, Unionfs detects that its branch configuration has changed and appropriately updates the file object.

Our current prototype of file-level revalidation is implemented at the entry point of each Unionfs file method to allow Unionfs to operate with an unmodified kernel. However, some simple system calls such as `fstat` read the file structure without first validating its contents. Ideally, the VFS should handle this functionality so that the service is exposed to all file systems.

5.4 On-Exit Callbacks

Linux does not provide a way for kernel components to register interest in the death of a process. We extended the Linux kernel to allow an arbitrary number of private data fields and corresponding callback functions to be added to the per-process data structure, `struct task`. When a process exits, the kernel first executes any callback functions before destroying the `task` object. NCryptfs uses such callbacks in two ways. First, when processes die, we immediately remove any active sessions that are no longer valid (e.g., if the last process in a session dies, we remove the NCryptfs active session). The alternative to on-exit callbacks would have been to use a separate kernel thread to perform periodic garbage collection, but that leaves security data vulnerable for an unacceptably long window. If an authenticated process terminates and the active session remains valid, then an attacker can quickly create many processes to hijack the active session entry. Using the on-exit callback, the information is invalidated before the process ID can be reused. Also with on-exit callbacks we release memory resources as soon as they are no longer needed.

The second use for private per-process data is in NCryptfs's challenge-response authentication ioctl, which proceeds as follows. First, the user process runs an ioctl to request a challenge. We generate a random challenge, store it as task-private data, and then send the challenge's size back to the user. Second, the user allocates a buffer and calls the ioctl again. This time the kernel actually returns the challenge data. The user performs some function on the data (e.g., HMAC-MD5 [33]), that requires some piece of secret knowledge to transform the data. Finally, the user program calls the ioctl a third time with the response. If the response matches what the kernel expects, then the user is authenticated. Using the task private data sets up a transaction between the kernel and a task. Even though there are several ioctls in this authentication sequence, it is no different than if a process had authenticated itself using a single ioctl. The challenge and its size are not useful to an attacker, since the challenge can only be used for a single authentication attempt. Only the last ioctl call modifies the state of the task. Finally, if an authentication is aborted, then the challenge is discarded on process termination.

Task private data has several uses outside of NCryptfs. In our Compound System Calls (CoSy) implementation there is a shared user-kernel buffer that enables zero-copy system calls [47–49]. Task private data is used to efficiently free this shared buffer on exit. Our Elastic Quota system also makes use of task-private data to store per-user policies [74, 75].

5.5 User-Space Callbacks

User-space callbacks allow the kernel to offload difficult functionality to user-space and provide for richer interaction with users. Presently, Linux invokes a user-space program, `modprobe`, to handle the task of loading kernel modules on demand. This offloads complicated parsing of configuration files and processing to user space.

We have integrated user-space callbacks with our file system to allow for enhanced usability. When a user sets a timeout in NCryptfs, they can also specify a user-space helper. As seen in Figure 5.2, this helper may, for example, prompt the user to reenter their key so that the operation can continue.

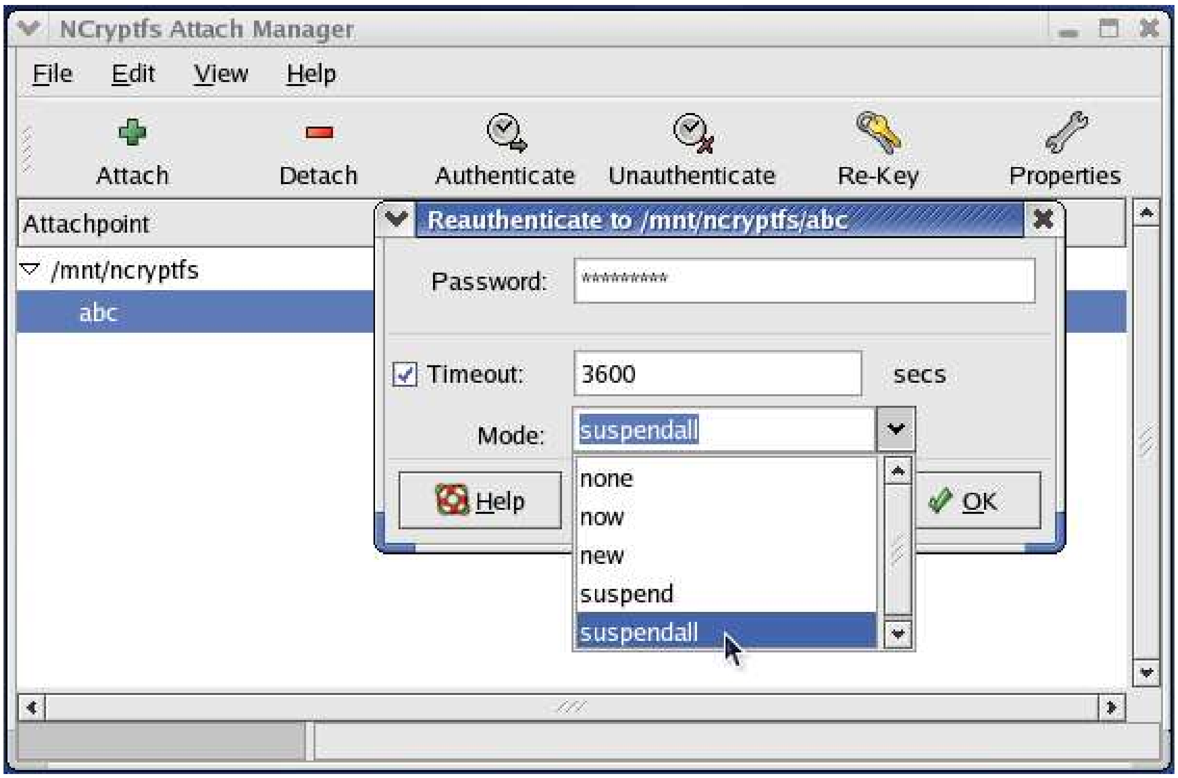


Figure 5.2: Using user-space callbacks, NC-Tools can be automatically called when a key timeout occurs.

The NCryptfs user-space callback code works similarly to the `modprobe` helper code. To execute a new helper, a new kernel thread is created. From inside the new kernel thread, the `exec` system call is used to replace the current process image and the user-mode helper begins to execute. Instead of executing the helper as root using the file system context of `init` (as is done with `modprobe`), NCryptfs uses the current user ID and the current process's file system context (i.e., root file system and working directory). We believe these types of callbacks from kernel to user space are essential to providing an easy-to-use interface for new file system functionality.

5.6 Bypassing VFS Permissions

NCryptfs supports native UNIX groups like any other entity. A UNIX group has some disadvantages, primarily that a group needs to be setup by the system administrator ahead of time. This means that users must contact the system administrator, and then wait for action to be taken.

NCryptfs supports ad-hoc groups by simply adding authorizations for several individual users (or other entities). The problem with this approach is that each additional user must have permissions to modify the lower level objects, since NCryptfs by default respects the standard lower-level file system checks. If the permissions on the lower level objects are relaxed, then new users can modify the files. However, without a corresponding UNIX group, it is difficult to give permissions to precisely the subset of users that must have them. If the permissions are too relaxed, then rogue users can trivially destroy the data and cryptanalysis become easier—even without the system being compromised. NCryptfs's bypass-VFS-permissions option solves this problem. The owner of the attach can delegate this permission (assuming root has given it to the owner). When this is enabled, NCryptfs performs all permission checks independently of the lower-level file system. This allows NCryptfs to be used for a wider range of applications

When intercepting permissions checks, it is trivial to implement a policy that is more restrictive than the underlying file system's normal UNIX mode-bit checks. To support ad-hoc groups without changing lower-level file systems, however, NCryptfs needed to completely ignore the lower-level file system's mode bits so that NCryptfs could implement its own authentication checks and yet appear to access the lower-level files as their owner.

The flow of an NCryptfs operation that must bypass VFS permissions (e.g., `unlink`) is as follows:

```
sys_unlink {                              /* system call service routine */
  vfs_unlink {                                           /* VFS method */
    call nc_permission()
    if not permitted: return error
    nc_unlink {                                     /* NCryptfs method */
      call nc_perm_preop()                           /* code we added */
      vfs_unlink {                                       /* VFS method */
        call ext2_permission()
        if not permitted: return error
        call ext2_unlink()                             /* EXT2 method */
      }                                       /* end of inner vfs_unlink */
      call nc_perm_fixup()                           /* code we added */
    }                                             /* end of nc_unlink */
  }                                           /* end of outer vfs_unlink */
}                                                /* end of sys_unlink */
```

The VFS operation (e.g., `vfs_unlink`) checks the permission using the `nc_permission` function. If the permission check succeeds, the corresponding NCryptfs-specific operation is called (e.g., `nc_unlink`). NCryptfs locates the lower-level object and again calls the VFS operation. The VFS operation checks permissions for the lower-level inode before calling the lower-level operation. This control flow means that we can not actually intercept the lower-level permission call. Instead, we change `current->fsuid` to the owner of the lower-level object before the operation is performed and restore it afterward, which is done in `nc_perm_preop` and `nc_perm_fixup`, respectively. We change only the permissions of the current task, and the process can not perform any additional operations until we restore the `current->fsuid` and return from the NCryptfs function. This ensures that only one lower file system operation can be performed between `nc_perm_preop` and `nc_perm_fixup`.

Using bypass VFS permissions, we have also implemented POSIX ACLs [19, 26] in a stackable file system. In this file system, ACLs are stored in a Berkeley Database [58] and can be layered over any other file system (e.g., Ext2 or VFAT).

Linux 2.6 will have Linux Security Modules (LSMs) that allow the interception of many security operations [68]. Unfortunately, the LSM framework is not sufficient to bypass lower-level permissions either. Their VFS calls the file-system–specific permission operation first. The LSM permissions operation is called only if the file-system–specific operation succeeds. The LSM operation can allow or deny access only to objects that the file system has already permitted. A better solution is to consult the file-system–specific permission operation. This result should be passed to the LSM module which can make the final decision, possibly based on the file-system–specific result.

Chapter 6

Conclusions and Future Work

The contributions of this work are that we have presented a survey of available file encryption and snapshotting systems. We have performed a comprehensive performance comparison of five representative encryption systems: Cryptoloop using a device as a backing store, Cryptoloop using a file as a backing store, EFS, CFS, and NCryptfs.

We have developed several operating system changes to support our work on secure file systems. Split-view caches allow different processes to have distinct views of the same namespace. File revalidation allows the file system to perform appropriate access checks and ensure that objects reflect true system state. On-exit callbacks close an attacker's window of opportunity by expunging authentication data precisely when it becomes invalid. When combined with task private data, on-exit callbacks are also useful for a wide variety of non-security applications. User-space callbacks allow rich new interaction between the user and the file system. The ability for a stackable file system to bypass VFS permissions allows us to layer useful new functionality, like ACLs, over existing file systems.

The rest of this Chapter is organized as follows. Section 6.1 discusses our conclusions about snapshotting and versioning systems. Section 6.2 discusses our conclusions about cryptographic file systems. We present possible future directions in Section 6.3.

6.1 Snapshotting and Versioning Systems

From our survey of snapshotting and versioning systems, and our own work, we believe that both have distinct roles. Snapshotting systems are an ideal augmentation to full system backups; snapshots allow the administrator to almost instantly capture the state of the file system at one point in time. On the other hand, versioning systems allow individual users to have control over their files and easily correct mistakes. For security, versioning systems give the administrator a more complete view of the system's changes, which is useful for forensics.

Most snapshotting systems require knowledge of both the block-level and the file system. Examples of these systems include FreeBSD Soft Updates, WAFL, and Ext3Cow. We believe block-level and file system snapshotting functionality should be separate, because snapshotting is a useful service that should not have to be re-implemented for each file system. Clotho changes only the block level, and allows any file system to be snapshotted [16]. Unionfs can layer snapshotting on top of existing file systems, without the knowledge of the block level or even over network file systems (e.g., NFS or CIFS) [20].

Versioning systems at the file system level are a useful for users. Disk-based file systems, like Elephant or CVFS, require replacing existing file systems. Stackable file systems can layer on top of existing systems, so we believe that they are the best candidate for most installations. If versions are needed for each modification (e.g., atime updates), then disk-based file systems are a good alternative to stackable file systems because they can employ specialized on-disk structures for efficient version storage.

User-level libraries and applications are cumbersome, and because they do not export a true file system interface can not be used with all applications or for security purposes.

6.2 Cryptographic File Systems

From our cryptographic file system performance study we draw five conclusions:

- It is often suggested that entire file systems should not be encrypted because of performance penalties. We believe that encrypting an entire hard disk is practical. Systems such as Cryptoloop that can encrypt an entire partition are able to make effective use of the buffer cache in order to prevent many encryption and decryption operations.

- For single process workloads, I/O is the limiting factor, but the encryption operations lie along a critical path in the I/O subsystem. Since I/O operations can not complete until the encryption/decryption is done, encryption negatively impacts performance in some cases.

- The loop device was able to scale along with the underlying file system for both metadata and I/O-intensive operations. NCryptfs was able to scale along with the file system for metadata operations, but not for I/O-intensive operations. The single threaded nature of CFS limits its scalability. EFS and NTFS did not perform well for the workload with many concurrent files, but EFS was able to exploit concurrent processes in both the Postmark and IOMeter test.

- The effect of caching can not be underestimated when comparing cryptographic file systems. Caches not only decrease the number of I/O operations, but they also avoid costly encryption operations. Even though Blowfish is almost twice as fast as AES on our machines, their results were quite similar, in large part because many I/O requests are served out of the cache.

- When heavily interleaving disk and CPU operations unexpected effects often occur. Furthermore, these effects are not necessarily gradual degradation of performance, but rather large spikes in the graph. Adjusting any number of things (memory, disk speed, or other kernel parameters) can expose or hide such effects. For performance-critical applications, the specific hardware, operating system, and file system must be benchmarked with both the expected workload as well as lighter and heavier workloads.

Based on our study, we believe block devices are a promising way to encrypt data for installations that do not need the advanced features that can only be provided by a file system (e.g., ad-hoc groups or per-file permissions). However, there are still two major problems with the state of block-based encryption systems. First, loop devices using a file as a backing store are not currently opened with `O_SYNC`. This means that even though the file system contained in the encrypted volume flushes data to the block device, the underlying file system does not flush the data. This interaction increases performance (even when compared to the native file system) at the expense of reliability and integrity constraints. This should be changed in future releases of Cryptoloop. NCryptfs and CFS also do not respect the `O_SYNC` flag, but for different reasons. NCryptfs does not because of the VFS call path, and CFS simply uses all asynchronous I/O.

Second, there are a large number of block-based encryption systems, for the same and different operating systems, but they are not compatible due to different block sizes, IV calculation methods, key derivation techniques, and disk layouts. For these systems to gain more wide-spread use they should support a common on-disk format. A common format would be particularly useful for removable media (e.g., USB drives).

Stackable or disk-based file systems are useful when more advanced features are required. The following three issues should be addressed. First, the write-through implementation used by FiST results in more encryption operations than are required. A write-back implementation would allow writes to be coalesced in the upper-level file system. Second, the encoding size of NCryptfs should be smaller than PAGE_SIZE in order to reduce computational overhead. Third, the FiST stackable file system templates should have a buffer cache mode. Presently, FiST uses only the page cache, but to ensure good performance the Linux VFS design relies on functionality provided by the buffer cache. Also, the Linux VFS is designed in such a way that file systems that do not use the buffer cache do not have their writes throttled and are more prone to deadlock. Unfortunately, the buffer cache design relies upon device drivers. In our file system built on top of the Berkeley Database, we use a buffer cache design with a pseudo-device driver. We will take a similar approach to improve our FiST templates.

6.3 Future Work

Confidentiality can currently be protected through cryptographic file systems, but most systems do not yet properly ensure the integrity of data. Using tweakable block ciphers, integrity is improved because changes are more noticeable and cut-and-paste attacks are not possible. However, without increasing the size of the device, it is impossible to automatically detect and be assured that the data read from the device was the data that was written to the device. Both a file system or loopback device that stores cryptographic checksums of data would greatly increase the overall usefulness of cryptographic file systems. Many storage solutions already use extra space to improve availability in the form of RAID, so we believe that the extra space for cryptographic checksums to assure integrity would be well worth it. We will investigate the performance impact of checksumming, and the most efficient place to store these checksums (interspersed with data or in a separate location).

Memory-mapped operations are efficient because they allow applications to access data with fewer context switches into the kernel. This performance advantage does however complicate secure file system design, because data is accessed without the file system's knowledge. We will develop a new API for file systems to validate `mmap`ed accesses to their data. Since this will likely have a large impact on `mmap` performance, we will design our API to be flexible: a file system may only want to validate reads or writes, or there may be an acceptable window between accesses (e.g., if a process is granted access once, then it remains authorized for the next second). This API will allow improved snapshotting and improved authentication checks for cryptographic file systems. Since the page fault handler will be more flexible, this API will also allow new functionality aside from security applications (e.g., compressed memory).

The virtual file system should include a centralized, stacking-aware cache manager. This would have several benefits. First, file systems that do not manipulate file data (e.g., Unionfs or Tracefs) could keep a single copy of data pages in the cache and still intercept data manipulation operations. This would improve both performance and flexibility. A stackable file system that filters data (e.g., NCryptfs) could inform the lower level system that caching this data is not useful. For NCryptfs, eliminating double-buffering would decrease encryption and I/O operations. Heidemann developed a unified cache manager for stackable file systems, but his solution only considered the page cache; and considered only a single linear stack [23]. We will design and build a more flexible and efficient unified caching system for all file system data (data pages, inodes, directory entries, etc.), which also considers the important class of fanout file systems.

Bibliography

[1] IOMeter. http://iometer.sourceforge.net, August 2003.

[2] AIM Technology. AIM Multiuser Benchmark - Suite VII Version 1.1. http://sourceforge.net/projects/aimbench, 2001.

[3] Backup Direct. Value of business data. www.backupdirect.net/about_situation_today.htm, 2004.

[4] B. Berliner and J. Polk. Concurrent Versions System (CVS). www.cvshome.org, 2001.

[5] BitMover. Bitkeeper – the scalable distributed software configuration management system. www.bitkeeper.com, 2004.

[6] M. Blaze. A cryptographic file system for Unix. In *Proceedings of the first ACM Conference on Computer and Communications Security*, 1993.

[7] R. Bryant, D. Raddatz, and R. Sunshine. PenguinoMeter: A New File-I/O Benchmark for Linux. In *Proceedings of the Fifth Annual Linux Showcase & Conference*, pages 5–10, Oakland, CA, November 2001.

[8] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 245–252, June 2001.

[9] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 43–60, San Francisco, CA, 1992.

[10] CollabNet, Inc. Subversion. http://subversion.tigris.org, 2004.

[11] M. Corner and B. D. Noble. Zero-interaction authentication. In *The Eigth ACM Conference on Mobile Computing and Networking*, September 2002.

[12] Department of Commerce: National Institute of Standards and Technology. Announcing Development of a Federal Information Processing Standard for Advanced Encryption Standard. Technical Report Docket No. 960924272-6272-01, January 1997.

[13] R. Dowdeswell and J. Ioannidis. The CryptoGraphic Disk Driver. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, June 2003.

[14] D. Ellard and M. Seltzer. NFS Tricks and Benchmarking Traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 101–114, June 2003.

[15] First Backup. First backup product overview. www.firstbackup.com/Product, 2004.

[16] M. D. Flouris and A. Bilas. Clotho: Transparent Data Versioning at the Block I/O Level. In *Proceedings of the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST 2004)*, pages 315–328, April 2004.

[17] G. Fowler, D. Korn, S. North, H. Rao, and K. Vo. *Practical Reusable Unix Software*, chapter 2: Libraries and File System Architecture. J. Wiley & Sons, Inc., 1995.

[18] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar File System. *Communications of the ACM*, 31(3):288–298, 1988.

[19] A. Grüenbacher. Extended attributes and access control lists. http://acl.bestbits.at, 2002.

[20] P. Gupta, H. Krishnan, C. P. Wright, M. Zubair, J. Dave, and E. Zadok. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical Report FSL-04-01, Computer Science Department, Stony Brook University, January 2004. www.fsl.cs.sunysb.edu/docs/unionfs-tr/unionfs.pdf.

[21] P. C. Gutmann. Secure filesystem (SFS) for DOS/Windows. www.cs.auckland.ac.nz/~pgut001/sfs/index.html, 1994.

[22] S. Halevi and P. Rogaway. A tweakable enciphering mode. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference*, volume 2729 of *Lecture Notes in Computer Science*, pages 428–499, Santa Barbara, CA, August 2003. Springer.

[23] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. In *Proceedings of Fifteenth ACM Symposium on Operating Systems Principles*. ACM SIGOPS, 1995.

[24] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, pages 235–245, January 1994.

[25] IBM Rational. Rational clearcase. www.rational.com/products/clearcase/index.jsp.

[26] IEEE/ANSI. Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API)—Amendment: Protection, Audit, and Control Interfaces [C Language]. Technical Report STD-1003.1e draft standard 17, ISO/IEC, October 1997. *Draft was withdrawn in 1997.*

[27] Jetico, Inc. BestCrypt software home page. www.jetico.com, 2002.

[28] P. H. Kamp. *gdbe(4)*, October 2002. FreeBSD Kernel Interfaces Manual, Section 4.

[29] J. Katcher. PostMark: a New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. www.netapp.com/tech_library/3022.html.

[30] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 59–72, San Francisco, CA, March/April 2004.

[31] W. Koch. The GNU privacy guard. www.gnupg.org, August 2003.

[32] D. G. Korn and E. Krell. The 3-D File System. In *Proceedings of the USENIX Summer Conference*, pages 147–156, Summer 1989.

[33] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. Technical Report RFC 2104, Internet Activities Board, February 1997.

[34] E. Mauriello. TCFS: Transparent Cryptographic File System. *Linux Journal*, (40), August 1997.

[35] A. D. McDonald and M. G. Kuhn. StegFS: A Steganographic File System for Linux. In *Information Hiding*, pages 462–477, 1999.

[36] M. K. McKusick and G. R. Ganger. Soft Updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 1–18, JUNE 1999.

[37] Microsoft Corporation. Encrypting File System for Windows 2000. Technical report, July 1999. `www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp`.

[38] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 115–128, San Francisco, CA, March/April 2004.

[39] R. Nagar. *Windows NT File System Internals: A developer's Guide*, pages 615–67. O'Reilly, September 1997. Section: Filter Drivers.

[40] National Institute of Standards and Technology. *FIPS PUB 46-3: Data Encryption Standard (DES)*. National Institute for Standards and Technology, Gaithersburg, MD, USA, October 1999.

[41] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback. Report on the Development of the Advanced Encryption Standard (AES). Technical report, Department of Commerce: National Institute of Standards and Technology, October 2000.

[42] J. Nugent, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Controlling Your PLACE in the File System with Gray-box Techniques. In *Proceedings of the Annual USENIX Technical Conference*, pages 311–323, June 2003.

[43] J. S. Pendry and M. K. McKusick. Union mounts in 4.4BSD-Lite. In *Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems*, pages 25–33, December 1995.

[44] Z. N. J. Peterson and R. C. Burns. Ext3cow: The design, Implementation, and Analysis of Metadat for a Time-Shifting File System. Technical Report HSSL-2003-03, Computer Science Department, The Johns Hopkins University, 2003. `http://hssl.cs.jhu.edu/papers/peterson-ext3cow03.pdf`.

[45] R. Power. Computer Crime and Security Survey. *Computer Security Institute*, VIII(1):1–24, 2002. `www.gocsi.com/press/20020407.html`.

[46] The OpenSSL Project. Openssl: The open source toolkit for SSL/TLS. `www.openssl.org`, April 2003.

[47] A. Purohit. A System for Improving Application Performance Through System Call Composition. Master's thesis, Stony Brook University, June 2003. Technical Report FSL-03-01, `www.fsl.cs.sunysb.edu/docs/amit-ms-thesis/cosy.pdf`.

[48] A. Purohit, J. Spadavecchia, C. Wright, and E. Zadok. Improving Application Performance Through System Call Composition. Technical Report FSL-02-01, Computer Science Department, Stony Brook University, June 2003. `www.fsl.cs.sunysb.edu/docs/cosy-perf/cosy.pdf`.

[49] A. Purohit, C. Wright, J. Spadavecchia, and E. Zadok. Develop in User-Land, Run in Kernel Mode. In *Proceedings of the 2003 ACM Workshop on Hot Topics in Operating Systems (HotOS IX)*, May 2003.

[50] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of First USENIX conference on File and Storage Technologies*, pages 89–101, January 2002.

[51] R. Richardson. Computer Crime and Security Survey. *Computer Security Institute*, VIII(1):1–21, 2003. `www.gocsi.com/press/20030528.html`.

[52] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 15–30, Monterey, CA, January 2002.

[53] H. V. Riedel. The GNU/Linux CryptoAPI site. `www.kerneli.org`, August 2003.

[54] RSA Laboratories. Password-Based Cryptography Standard. Technical Report PKCS #5, RSA Data Security, March 1999.

[55] J. H. Saltzer. Hazards of file encryption. Technical report, 1981. `http://web.mit.edu/afs/athena.mit.edu/user/other/a/Saltzer/www/publications/csrrfc208.html`.

[56] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R.W. Carton, and J. Ofir. Deciding When to Forget in the Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 110–123, December 1999.

[57] B. Schneier. *Applied Cryptography*. John Wiley & Sons, second edition, October 1995.

[58] M. Seltzer and O. Yigit. A new hashing package for UNIX. In *Proceedings of the Winter USENIX Technical Conference*, pages 173–84, January 1991. `www.sleepycat.com`.

[59] J. Shapiro. OpenCM. `www.opencm.org`, 2004.

[60] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS Version 4 Protocol. Technical Report RFC 3010, Network Working Group, December 2000.

[61] D. A. Solomon and M. E. Russinovich. *Inside Microsoft Windows 2000*, chapter 12: File Systems, pages 683–778. Microsoft Press, 2000.

[62] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata Efficiency in Versioning File Systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies*, pages 43–58, March 2003.

[63] J. Spadavecchia and E. Zadok. Enhancing NFS Cross-Administrative Domain Access. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 181–194, June 2002.

[64] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 165–180, October 2000.

[65] D. Tang and M. Seltzer. Lies, Damned Lies, and File System Benchmarks. Technical Report TR-34-94, Harvard University, December 1994. In VINO: The 1994 Fall Harvest.

[66] VERITAS Software. VERITAS File Server Edition Performance Brief: A PostMark 1.11 Benchmark Comparison. Technical report, Veritas Software Corporation, June 1999. `http://eval.veritas.com/webfiles/docs/fsedition-postmark.pdf`.

[67] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proceedings of the Eigth Usenix Security Symposium*, August 1999.

[68] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[69] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, June 2003.

[70] E. Zadok. *Linux NFS and Automounter Administration*. Sybex, Inc., May 2001.

[71] E. Zadok, J. M. Anderson, I. Bădulescu, and J. Nieh. Fast Indexing: Support for size-changing algorithms in stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 289–304, June 2001.

[72] E. Zadok, I. Bădulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998. `www.cs.columbia.edu/~library`.

[73] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, June 2000.

[74] E. Zadok, J. Osborn, A. Shater, C. P. Wright, K. Muniswamy-Reddy, and J. Nieh. Reducing Storage Management Costs via Informed User-Based Policies. Technical Report FSL-02-02, Computer Science Department, Stony Brook University, September 2002. `www.fsl.cs.sunysb.edu/docs/equota-policy/policy.pdf`.

[75] E. Zadok, J. Osborn, A. Shater, C. P. Wright, K. Muniswamy-Reddy, and J. Nieh. Reducing Storage Management Costs via Informed User-Based Policies. In *Proceedings of the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST 2004)*, pages 193–197, April 2004.